



Professional Bachelor Applied Information Technology



Designing Scalable Web Applications with Node.js and PostgreSQL: Best Practices for Caching and Load Balancing

Jonathan Claessens

Promoters:

Johan Maes
Tom Schuyten

Lemonade
PXL University College Hasselt



Bachelor paper Academic year 2022-2023

Acknowledgements

I would like to express my sincere gratitude to my internship mentor, Johan Maes, and Lemonade for giving me the opportunity to conduct my research in their organization. Their guidance, support, and valuable feedback have been instrumental in the success of this research project.

I would also like to extend my appreciation to my colleagues at Lemonade, Carlos Navidad, Stefan Van der Straeten, David Sanchez, and Matthias Snellings, who provided me with valuable advice and assistance throughout the internship. Their contributions were essential to the success of this project, and I am grateful for their help.

Furthermore, I want to thank PXL for organizing and including internships in their study trajectory, providing students with opportunities to gain real-world experience. It has been an honor to study at this academic institution.

Lastly, I want to express my heartfelt gratitude to my parents for their continued support during my entire education. It would not have been possible to have done this without their unwavering support.

To all of you, a big thank you.

Abstract

Lemonade is a multinational IT company headquartered in Leuven, Belgium, with offices in Barcelona, Buenos Aires, and Stockholm. The company specializes in customized solutions for web development, e-commerce, and digital marketing, using agile methodologies and CI/CD practices. With this in mind, the company has identified the need to update and improve the existing mail user interface used by the educational organization 'Katholiek Onderwijs Vlaanderen'.

The organization's Angular.js application has become outdated and requires enhancements to improve its performance. This project focuses on converting the mail user interface from Angular.js to React, which is a more efficient technology for web application development. The project involves the enhancement of the application's overall performance by optimizing its code and implementing changes to reduce loading times and improve responsiveness for the end-users.

By undertaking this project, Lemonade aims to demonstrate its commitment to providing innovative and high-quality solutions to its clients. The successful completion of this project aims to provide a better user experience for the educational organization's end-users and enhance Lemonade's reputation as a leading provider of technological solutions.

The project requires the use of various technologies, including but not limited to Angular.js, React, BitBucket, Monday, and Jenkins. The primary emphasis is on the frontend development aspect of the project, with involvement in backend development as well.

This thesis aims to explore the best practices for designing scalable web applications using Node.js and PostgreSQL, focusing on caching and load balancing. It discusses the concepts of caching and load balancing and their importance in scaling web applications. In addition, it explores various caching techniques and tools that can be utilized with Node.js and PostgreSQL. Finally, the thesis provides guidelines for implementing caching and load balancing in a Node.js and PostgreSQL web application and evaluates their effectiveness in improving the application's scalability.

List of tables

Acknowledgements	ii
Abstract	iii
List of tables	iv
List of figures	vi
List of abbreviations	ix
Glossary	1
Introduction.....	5
I. Traineeship report.....	6
1 About the company.....	6
1.1 Lemonade	6
1.2 Context internship.....	6
1.3 Objective internship	6
1.4 Working environment	7
1.4.1 Team	7
1.4.2 Process and implementation.....	7
1.4.3 Infrastructure and technologies	8
2 Elaboration of the assignment start.....	11
2.1 Introduction: from Angular.js to React	11
2.2 Elaboration of the assignment	13
2.3 Conclusion of the internship assignment.....	14
II. Research topic	24
1 Research questions, sub-questions, and hypotheses	24
1.1 Research purpose	24
1.2 The main research question is:	24
1.3 The sub-questions that need to be answered are:	24
2 Research method	26
3 Research elaboration	27
3.1 Literature review	27
3.1.1 What is Node.js?.....	27
3.1.2 What is PostgreSQL?	30
3.1.3 What is caching and its importance for web scalability?	31
3.1.4 Why is load balancing important for web scalability?	33
3.1.5 Best practices for caching in Node.js and PostgreSQL web applications?	35

3.1.6	Real-world examples of successful Node.js and PostgreSQL web applications with caching and load balancing?.....	41
3.1.7	Challenges and trade-offs in implementing caching and load balancing in Node.js and PostgreSQL web applications?	42
3.2	Proof of Concept.....	45
3.2.1	High level overview	45
3.2.2	Set up Node.js & PostgreSQL project	45
3.2.3	Set up caching.....	52
3.2.4	Set up Siege	54
3.2.5	Load testing caching	55
3.2.6	Set up load balancing	59
3.2.7	Load testing balancing.....	61
	Conclusion	65
III.	Reflection.....	66
4	Bibliographical references	68

List of figures

Figure 1: Lemonade logo	6
Figure 2: Agile	8
Figure 3: BitBucket logo.....	8
Figure 4: Monday logo.....	9
Figure 5: Jenkins	9
Figure 6: Heroku logo	10
Figure 7: Angular.js logo	11
Figure 8: React.js logo	11
Figure 9: Angular.js application UI	13
Figure 10: Angular.js application UI with data	14
Figure 11: Angular.js application responsive.....	14
Figure 12: ticket 1	14
Figure 13: ticket 2.....	15
Figure 14: application header.....	15
Figure 15: application user dropdown	15
Figure 16: application user with picture dropdown.....	16
Figure 17: ticket 3.....	16
Figure 18: application mail display.....	16
Figure 19: application status display	17
Figure 20: application action button click result.....	17
Figure 21: ticket 4.....	17
Figure 22: application filter	18
Figure 23: application date filter	18
Figure 24: application component filter.....	19
Figure 25: ticket 5.....	19
Figure 26: application load more button	20
Figure 27: ticket 6.....	20
Figure 28: application query parameters	20
Figure 29: ticket 7	20
Figure 30: application impersonation success	21
Figure 31: application impersonation failed	21
Figure 32: ticket 8.....	21
Figure 33: application loading message	22
Figure 34: application dropdown css fix.....	22
Figure 35: finished react application	22
Figure 36: finished react application responsive.....	23
Figure 37: Node.js code example	27
Figure 38: synchronous file read example	29
Figure 39: asynchronous file read example	29
Figure 40: Synchronous file read example expanded	29
Figure 41: Asynchronous file read example expanded	29
Figure 42: load balancing diagram	34
Figure 43: cache-aside concept.....	36
Figure 44: read-through concept	37
Figure 45: write-through concept	37
Figure 46: write-back concept.....	38

Figure 47: refresh-ahead concept	39
Figure 48: proof of concept overview	45
Figure 49: PostgreSQL CMD	45
Figure 50: PostgreSQL CMD	45
Figure 51: PostgreSQL CMD	46
Figure 52: PostgreSQL CMD	46
Figure 53: CMD to setup project	46
Figure 54: CMD to install dependency	46
Figure 55: Setup project Node.js	47
Figure 56: New express path code	47
Figure 57: Finish setup project express	47
Figure 58: CMD to run project.....	47
Figure 59: Retrieve data example	47
Figure 60: Configuration database	48
Figure 61: Retrieve all users code	48
Figure 62: Retrieve user by id code	49
Figure 63: Insert into users code	49
Figure 64: Update user code	49
Figure 65: Delete user code.....	50
Figure 66: Export functions code.....	50
Figure 67: Import code	50
Figure 68: Setup endpoints code.....	50
Figure 69: CMD to run the project	51
Figure 70: Postman GET request example	51
Figure 71: Postman add user example	51
Figure 72: Postman add user output.....	52
Figure 73: Install dependency	52
Figure 74: CMD to install Redis	52
Figure 75: CMD to confirm installation of Redis	52
Figure 76: CMD to copy Redis to path.....	52
Figure 77: CMD to execute the Redis server	52
Figure 78: CMD to indicate Redis is running successfully	53
Figure 79: Ping output example	53
Figure 80: Import Redis code	53
Figure 81: Implement caching on endpoint code	53
Figure 82: GET request example	54
Figure 83: GET request example repeated.....	54
Figure 84: install siege CMD	54
Figure 85: Siege test CMD	54
Figure 86: content file parameter	56
Figure 87: first test with caching	56
Figure 88: first test without caching and load balancing	56
Figure 89: second test with caching.....	57
Figure 90: second test without caching and load balancing	57
Figure 91: third test with caching.....	57
Figure 92: third test without caching and load balancing.....	58
Figure 93: Nginx configuration file	59
Figure 94: Setup load balancing	60

Figure 95: CMD to run Nginx	60
Figure 96: Get requests example	61
Figure 97: first with load balancing	61
Figure 98: first test without caching and load balancing	62
Figure 99: second test with load balancing.....	62
Figure 100: second test without caching and load balancing	62
Figure 101: third test with load balancing.....	62
Figure 102: third test without caching and load balancing.....	63

List of abbreviations

Table 1 – List of used abbreviations

Abbreviations	Explanation
I/O	Input/Output
ORM	Object Relational Mapping
CDN	Content Delivery Network
UI	User Interface
HQ	Headquarters
IT	Information Technology
API	Application Programming Interface
UX	User Experience
DOM	Document Object Model
CSS	Cascading Style Sheets
OAuth	Open Authorization
OS	Operating System
CPU	Central processing unit
Fs	File system
SQL	Structured Query Language
OLTP	Online Transaction Processing
DDoS	Distributed Denial of Service
SSL	Secure Sockets Layer
RDS	Remote Dictionary Server
HTTP	Hypertext Transfer Protocol
CMD	Command
DB	Database
PCI	Payment Card Industry
JSON	JavaScript Object Notation
CRUD	Create, Read, Update, and Delete
WSL	Windows Subsystem for Linux
AWS	Amazon Web Services
LRU	Least Recently Used
LFU	Least Frequently Used
HTML	HyperText Markup Language
ACID	Atomicity Consistency Isolation Durability
MVCC	Multiversion Concurrency Control
RAM	Random Access Memory
TTL	Time-to-live
MRU	Most Recently Used
FIFO	First In, First Out
TLS	Transport Layer Security
URL	Uniform Resource Locator
POC	Proof Of Concept
CRUD	Create, Read, Update, Delete
WSL	Windows Subsystem for Linux

Glossary

Table 2 – List of used concepts

Concept	Meaning
Callback	<i>A callback is a function that is passed as an argument to another function and is called when the first function has completed its task.</i>
Object Relational Mapping	<i>A technique used in computer programming to map between objects in an object-oriented programming language and tables in a relational database management system.</i>
Scrum	<i>An agile methodology used for software development.</i>
V8	<i>Google's open-source high-performance JavaScript engine.</i>
User Story	<i>A brief description of a feature or functionality of a software application, written from the perspective of the user.</i>
Workflow	<i>A series of steps or tasks that need to be completed in order to achieve a particular outcome.</i>
Component	<i>A reusable piece of code that represents a specific part of the user interface. Components typically consist of HTML, CSS, and JavaScript code, and can be thought of as "building blocks" that can be combined to create more complex user interfaces.</i>
Library	<i>A collection of code that provides pre-written functionality for specific programming tasks. Libraries are typically used by developers to save time and reduce the amount of code they need to write.</i>
Framework	<i>A set of pre-written code that provides a structure for developing applications.</i>
Callback function	<i>A callback function is a function that is passed as an argument to another function and is called back after the first function completes its execution. The purpose of a callback function is to allow asynchronous programming and to handle events or data that are not available immediately.</i>
Throughput	<i>The amount of data or work that can be processed by a system or a network in a given period of time. It is a measure of the system's or network's performance and efficiency. In computing, throughput is often used to measure the performance of networks, storage devices, and other hardware components.</i>
Thread	<i>A thread is a single sequence of instructions that can be executed independently within a program. Threads are used to enable concurrent execution of multiple tasks within a program, allowing for greater efficiency and performance.</i>
Database	<i>A collection of organized data that can be accessed, managed, and updated easily.</i>
Omnipresent	<i>Present everywhere at the same time.</i>
Overhead	<i>Additional resources or work required to perform a particular task beyond the minimum required. In software development, overhead can refer to the additional time, cost, or resources required for a particular feature or process beyond what is necessary to complete the project.</i>
Static content	<i>Elements of a website or application that do not change</i>

	<i>frequently.</i>
Proxy server	<i>An intermediary server between a client and the internet.</i>
Payment Card Industry	<i>The standards and guidelines established by major credit card companies to ensure the secure handling of credit card information by merchants, service providers, and other entities involved in payment processing.</i>
Authentication	<i>The process of verifying the identity of a user or system.</i>
Session hijacking	<i>Session hijacking is an attack in which an attacker takes over a valid session between a user and a web application. The attacker can then use this session to impersonate the user and perform actions on their behalf.</i>
Cross-site scripting	<i>Cross-site scripting is a vulnerability that allows an attacker to inject malicious code into a web page viewed by other users. This can allow the attacker to steal user credentials or perform other malicious actions.</i>
Encryption	<i>The process of converting plain text into an encoded or encrypted format to secure it from unauthorized access or use.</i>
Authorization	<i>The process of granting or denying access to a resource or service based on the identity and permissions of the user or system requesting access.</i>
Digital marketing	<i>The strategic use of online channels and technologies to promote products, services, or brands and reach a targeted audience.</i>
Agile	<i>An iterative and flexible approach to project management that focuses on delivering value through adaptive planning, teamwork, and continuous feedback.</i>
Kanban	<i>A visual workflow management method that emphasizes continuous flow, transparency, and limiting work in progress to optimize efficiency and productivity.</i>
Pull request	<i>A collaborative process in software development where code changes made in a separate branch are proposed and reviewed before merging into the main codebase.</i>
Scaling	<i>The process of increasing or adjusting the capacity and capabilities of a system or infrastructure to handle higher workloads, increased demand, or accommodate growth.</i>
Deployment	<i>The process of releasing and making a software application available for use, typically involving the installation, configuration, and activation of the application on target systems or environments.</i>
Infrastructure	<i>The underlying foundation and resources, including hardware, software, and network components, that support the operation and functionality of a system or application.</i>
Server provisioning	<i>The process of setting up and configuring server resources to enable their operation and functionality.</i>
Robust	<i>Having a strong and resilient design or implementation that can handle various situations and challenges effectively.</i>
Document Object Model	<i>A programming interface that represents and interacts with the structure of an HTML or XML document.</i>
Client-side rendering	<i>The approach of rendering web content on the client's device using JavaScript, allowing for dynamic and interactive user experiences.</i>

Server-side rendering	<i>The process of generating HTML on the server and sending it to the client, enhancing performance and search engine visibility.</i>
Search engine optimization	<i>The practice of optimizing a website to improve its visibility and ranking in search engine results.</i>
Codebase	<i>The entire collection of source code files and resources that make up a software project.</i>
Vite	<i>A modern build tool that offers fast development server and optimized production builds for web applications.</i>
State management	<i>The management and control of application state, ensuring data consistency and synchronization across components.</i>
Hook	<i>A mechanism that allows developers to extend or modify the behavior of an API or software application.</i>
Dead-locking	<i>A situation in concurrent programming where two or more processes are unable to proceed because each is waiting for the other to release a resource.</i>
Kernel	<i>The core component of an operating system that provides essential services and manages system resources.</i>
Open-source	<i>Software that is released with a license allowing anyone to view, modify, and distribute the source code.</i>
Concurrency	<i>The ability of a system to execute multiple tasks or processes simultaneously.</i>
Atomicity Consistency Isolation Durability	<i>A set of properties that guarantee reliable and consistent transaction processing in a database system.</i>
Integrity	<i>The assurance of maintaining data accuracy, consistency, and reliability.</i>
Data corruption	<i>The unintended alteration or damage to data, resulting in its partial or complete loss of usability.</i>
Cache	<i>A temporary storage location that stores frequently accessed data to improve retrieval speed.</i>
Omnipresent	<i>Being present everywhere at the same time, often used to describe widespread or pervasive technology.</i>
Network latency	<i>The time delay experienced in network communications due to factors like distance and congestion.</i>
Compression	<i>The process of reducing the size of data to optimize storage or transmission efficiency.</i>
Cache cluster	<i>A group of interconnected cache servers that work together to enhance performance and reliability.</i>
Downstream	<i>Referring to the flow of data or events from a source to a destination or subsequent stages.</i>
Cassandra	<i>A highly scalable and distributed NoSQL database designed to handle large volumes of structured and unstructured data.</i>
Transport Layer Security	<i>A cryptographic protocol that ensures secure communication over a network.</i>
Cross-site scripting	<i>A security vulnerability where malicious scripts are injected into web pages viewed by users.</i>
Session hijacking	<i>Unauthorized access to a user's session by intercepting or stealing session information.</i>
Redis	<i>An in-memory data store that can be used as a database, cache, or message broker.</i>
Latency	<i>The time delay or lag experienced in transmitting data or</i>

	<i>performing operations.</i>
Content delivery network	<i>A geographically distributed network of servers that delivers web content to users based on their location.</i>
Reverse proxy	<i>A server that acts as an intermediary between client devices and web servers, providing additional functionality like caching and load balancing.</i>

Introduction

This thesis consists of two primary components: a research section and an elaboration of the internship assignment.

Research

This paper presents a comprehensive analysis of the scaling strategies for web applications, specifically examining the utilization of caching and load balancing techniques. The primary aim of this research is to assess the advantages and obstacles associated with the implementation of caching and load balancing within a Node.js and PostgreSQL web application.

The investigation focuses on elucidating the fundamental principles of caching and load balancing, while also evaluating various methodologies and tools that are well-suited for the aforementioned technologies. The ultimate objective is to establish practical guidelines for effectively integrating caching and load balancing mechanisms into web applications, emphasizing their role in enhancing scalability.

Internship assignment

The objective of this project is to migrate an Angular.js-based mail application to React, aiming to enhance its user interface and overall efficiency. The project entails the development of a new React-based UI, while retaining the existing API utilized by the Angular.js UI. The primary focus is on improving user-friendliness and performance for end-users, encompassing both frontend and backend development aspects.

I. Traineeship report

1 About the company

1.1 Lemonade



Figure 1: Lemonade logo

The company Lemonade, which was acquired by the multinational corporation Sword Group, has demonstrated a global presence through its establishment of offices in Leuven (Belgium), Barcelona (Spain), Buenos Aires (Argentina), and Stockholm (Sweden). The expansion of Lemonade's operations beyond its original location and its acquisition by a multinational company signify the company's growing reach, while still maintaining its headquarters in Leuven.

Lemonade specializes in offering a diverse range of IT services, including web development and digital marketing. While its primary focus lies within the Belgian and European markets, Lemonade has also effectively collaborated with clients from various regions around the world, highlighting its international engagement and capabilities.

The acquisition of Lemonade by Sword Group, a recognized multinational company, further enhances the significance and influence of Lemonade within the industry. This strategic move by Sword Group positions Lemonade for continued growth and development on a global scale.

1.2 Context internship

The internship took place in Barcelona, Spain, where the task involved the migration of an established mail application from Angular.js to React. This mail application is currently utilized by 'Katholiek Onderwijs Vlaanderen,' an educational organization, to facilitate efficient communication among its staff members. The objective of the internship was to enhance the application's performance and user experience through the adoption of the React framework.

1.3 Objective internship

The primary aim of this project is to enhance the efficiency and user-friendliness of the existing mail application by transferring its user interface (UI) from Angular.js to React. As part of this effort, the existing API used in the Angular.js UI will be utilized to maintain the application's functionality.

The author's primary responsibility in this project is to develop the new React UI while ensuring compatibility with the existing API. This involves both front-end and back-end development tasks.

The migration from Angular.js to React is motivated by the need to improve the user experience of the application. React is known for its high performance and flexibility, which can help deliver a smoother and more responsive UI. The new UI will also be designed to be more intuitive and user-friendly, with a focus on key usability metrics such as task completion time and error rates.

Overall, this project aims to improve the usability and performance of the existing mail application through a careful redesign of its UI using React, while preserving its essential features through the existing API.

1.4 Working environment

As per the terms of the contractual agreement with Lemonade, the employing organization, the work arrangement necessitates on-site presence for two days per week, with the remaining workload permitted to be fulfilled remotely. This blended approach affords advantages inherent in both on-site and remote work modalities, encompassing access to office amenities such as supplementary monitors and adaptable desks, alongside the flexibility and convenience of working from a remote location as deemed necessary. In essence, this harmonized arrangement engenders mutual benefits for the individual and the organization, fostering enhanced productivity, work-life equilibrium, and sustained engagement with the employing entity.

1.4.1 Team

During the internship period, an individual project was assigned with the specific objective of transitioning the user interface (UI) of the mail application from Angular.js to React. The assigned task entailed working independently, separate from the team members, while maintaining active participation in regular team meetings and seeking guidance and assistance from colleagues as necessary. The project benefitted from the invaluable guidance and support provided by the project supervisor, Johan Maes, who offered expert advice and constructive feedback to navigate through technical and design challenges. Moreover, the contributions of colleagues, namely Carlos Navidad, Stefan Van der Straeten, David Sanchez, and Matthias Snellings, proved to be significant in refining the project and enhancing its overall quality. The collaborative and supportive environment fostered a conducive atmosphere, enabling the attainment of project objectives while facilitating the acquisition of valuable experience in software development and team collaboration.

1.4.2 Process and implementation

During the internship period, the development team implemented a tailored Scrum methodology for agile software development, aiming to improve efficiency. The team transitioned from a Kanban approach to a release-based model characterized by 6-week release cycles, which facilitated enhanced planning and ensured timely software delivery.

Within the Scrum framework, the team diligently conducted daily meetings known as "daily scrums" to synchronize efforts and maintain effective communication. These sessions provided a platform for team members to discuss progress, tasks completed, upcoming tasks, and any obstacles hindering progress. The regular and structured communication fostered a cohesive workflow, enabled early issue identification, and facilitated prompt resolution.

The supervisor, Johan Maes, played an integral role within the team by providing guidance, support, and expertise as needed. This contribution contributed to the team's growth and the intern's professional development. The supervisor's involvement in monitoring progress and making necessary adjustments further ensured the successful completion of the project.

The adoption of the tailored Scrum methodology, incorporating 6-week release cycles, proved beneficial in optimizing the team's development efforts. The daily scrums and the guidance provided by the supervisor effectively enhanced the team's productivity, resulting in timely software releases and overall project success.



Figure 2: Agile

1.4.3 Infrastructure and technologies

The project under consideration pertains to the development of a mail application, with a primary emphasis on frontend development. The project incorporates a diverse array of technologies to fulfill its objectives, encompassing Angular.js and React frameworks for frontend implementation, BitBucket for source code management, Monday for user story handling, and Jenkins for deployment purposes.

The focal point of the frontend development endeavors revolves around the migration of an Angular.js component to a React component. The existing Angular.js component serves as a reference model for the subsequent development of its React counterpart. To facilitate this process, access to the Angular.js shared components is leveraged, enabling a thorough analysis and subsequent translation into React shared components. Additionally, pre-existing shared components developed in React are also integrated into the project's framework.

While the current focus is directed towards frontend development, it is worth mentioning that the backend infrastructure has already been established utilizing the Node.js runtime environment. The backend architecture provides essential APIs to facilitate data retrieval and manipulation for the frontend. Significantly, a PostgreSQL database serves as the central repository for storing and managing project-related data. Further exploration into the role of Node.js within the project will be addressed in the subsequent research topic.

Regarding the deployment aspect, the backend infrastructure is deployed on the Heroku platform, while the frontend components are hosted on AWS S3 buckets. These deployment environments provide the necessary infrastructure and resources to ensure the seamless execution and accessibility of the developed application.

BitBucket



Figure 3: BitBucket logo

Bitbucket is a web-based hosting service for software development projects that use either the Mercurial or Git revision control systems. It provides a platform for teams to collaborate on code development by allowing them to share, review, and manage code repositories. Bitbucket offers features such as pull requests, issue tracking, and integration with other development tools like Jenkins for continuous integration and deployment. It is owned by Atlassian and is often used in combination with other Atlassian products, such as Jira and Confluence.

Monday



Figure 4: Monday logo

Monday is a cloud-based project management software that allows teams to manage their work using visual boards, charts, and calendars. It provides a user-friendly interface that simplifies the process of creating, assigning, and tracking tasks. Monday also offers customizable workflows and automation features to streamline project management processes. One of its main features is the ability to create and manage user stories, which are used to define requirements and track progress. The tool provides a centralized location for user stories, allowing team members to collaborate and ensure that everyone is on the same page.

Jenkins



Figure 5: Jenkins

Jenkins is an open-source automation server that facilitates the building, testing, and deployment of software projects. It allows teams to automate parts of the software development process, such as building and testing code changes, and can integrate with various tools and technologies to streamline the development workflow. Jenkins supports a variety of programming languages, build tools, and version control systems, and provides an extensible plugin architecture for customizing its functionality. Its main benefits include automation of repetitive tasks, reduction of manual errors, and faster feedback loops for developers.

The project under consideration employs an array of technologies, such as Angular.js and React for frontend development, BitBucket for source code management, Monday for user story handling, and Jenkins for deployment. The primary aim of the project is to create a mail application, with a focus on frontend development. The frontend development task comprises the conversion of an Angular.js component into a React component by leveraging shared components of both Angular.js and React.

Notably, the backend infrastructure has already been established using Node.js and provides the frontend with the necessary API for data retrieval. Further details regarding the role of Node.js will be elaborated upon in the forthcoming research topic. In essence, the project relies on an array of tools and technologies that enable effective collaboration, streamline workflows, and automate development tasks.

Heroku



Figure 6: Heroku logo

Heroku is a cloud-based platform renowned for its simplified and streamlined approach to deploying, managing, and scaling web applications. With the aim of reducing complexities associated with infrastructure setup and management, Heroku offers developers a seamless environment for application deployment.

Built upon the infrastructure of Amazon Web Services (AWS), Heroku distinguishes itself by providing a high-level abstraction layer that shields developers from the intricacies of underlying infrastructure. This abstraction empowers developers to concentrate on writing code and deploying applications, relieving them from concerns about server provisioning, load balancing, and scaling.

Heroku boasts compatibility with multiple widely-used programming languages and frameworks, including Ruby, Python, Java, and Node.js, among others. The platform offers a streamlined deployment process facilitated by its command-line interface (CLI) and integrations with popular version control systems like Git. Developers can effortlessly push their code to a Heroku remote repository, triggering the platform's automatic building, deployment, and management processes.

2 Elaboration of the assignment start

2.1 Introduction: from Angular.js to React

To properly understand the process of transforming an Angular.js application to React, it is essential to have a solid understanding of both technologies.



Figure 7: Angular.js logo

Angular.js is a JavaScript framework developed by Google with the primary purpose of constructing web applications. It adopts a component-based architecture, enabling the creation of reusable user interface (UI) components. Notably, Angular.js incorporates various essential functionalities, including routing, forms handling, and the ability to perform HTTP requests. To enhance the language capabilities and provide additional robustness, TypeScript, a superset of JavaScript, is employed in Angular.js.

TypeScript enriches the JavaScript language with a wide range of advanced features that empower developers to create robust and maintainable code in Angular.js projects. Its support for strong typing enables the detection of potential type errors during development, enhancing the reliability and stability of Angular.js applications. TypeScript also offers features like interfaces, classes, modules, and decorators, facilitating code organization and reusability. Additionally, it provides advanced tools for code analysis and refactoring, improving code quality and reducing debugging and maintenance efforts. By incorporating TypeScript, developers can leverage these features to enhance productivity, code organization, and long-term maintainability in Angular.js-based projects.

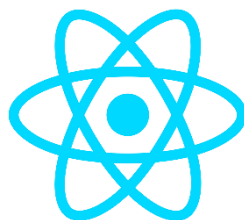


Figure 8: React.js logo

React, a JavaScript library developed by Facebook, serves as a powerful tool for constructing user interfaces. With React, developers can create reusable components that are rendered efficiently through the utilization of a virtual DOM, which selectively updates the relevant portions of a web page. This library is frequently employed alongside other frameworks and libraries, offering a declarative syntax that facilitates the specification of UI component structure and behavior.

Moreover, React extends its capabilities beyond client-side rendering by supporting server-side rendering, enabling the generation of HTML on the server before it is sent to the client. This feature enhances performance and facilitates search engine optimization. Furthermore, React enjoys widespread adoption and benefits from an active community of developers who contribute to its continuous improvement, provide support, and share valuable resources.

By leveraging React's virtual DOM, declarative syntax, server-side rendering capabilities, and active community, developers can create efficient and maintainable user interfaces for web applications. The combination of these features not only enhances development productivity but also promotes scalability and facilitates the creation of engaging user experiences.

There are several reasons why developers might consider switching an Angular.js application to React. Firstly, React has a more lightweight architecture compared to Angular.js. This means that React apps tend to load faster and perform better on slow devices or networks. React also uses a virtual DOM, which makes it faster to render changes to the UI, resulting in a smoother user experience.

Secondly, React's component-based architecture makes it easier to build complex UIs. Each component can be developed independently, making it simpler to manage the codebase and reducing the chances of code conflicts or bugs. Additionally, React provides a wide range of pre-built components and libraries, which can speed up development time.

Thirdly, React uses a more modern approach to web development, with a focus on functional programming and a declarative coding style. This makes the code easier to read and maintain, and also reduces the likelihood of errors. In contrast, Angular.js uses a more traditional, object-oriented approach, which can be more complex and harder to maintain.

Fourthly, React is highly flexible and can be used to build applications for multiple platforms, including web, mobile, and desktop. This makes it easier to create a consistent user experience across different devices and platforms.

The React community provides an invaluable support network for developers. It is characterized by its sizeable and active membership, offering continuous assistance, regular updates, and a wide range of resources. This collaborative environment enables developers to promptly address common challenges encountered in React development. Moreover, active participation in such a community ensures developers stay abreast of the latest advancements, improvements, and innovative practices within the React ecosystem.

While Angular.js has its own advantages, such as a more structured approach to development and stronger tooling support, the benefits of React make it an appealing option for many developers. Ultimately, the choice of framework depends on the specific needs and requirements of the project.

2.2 Elaboration of the assignment

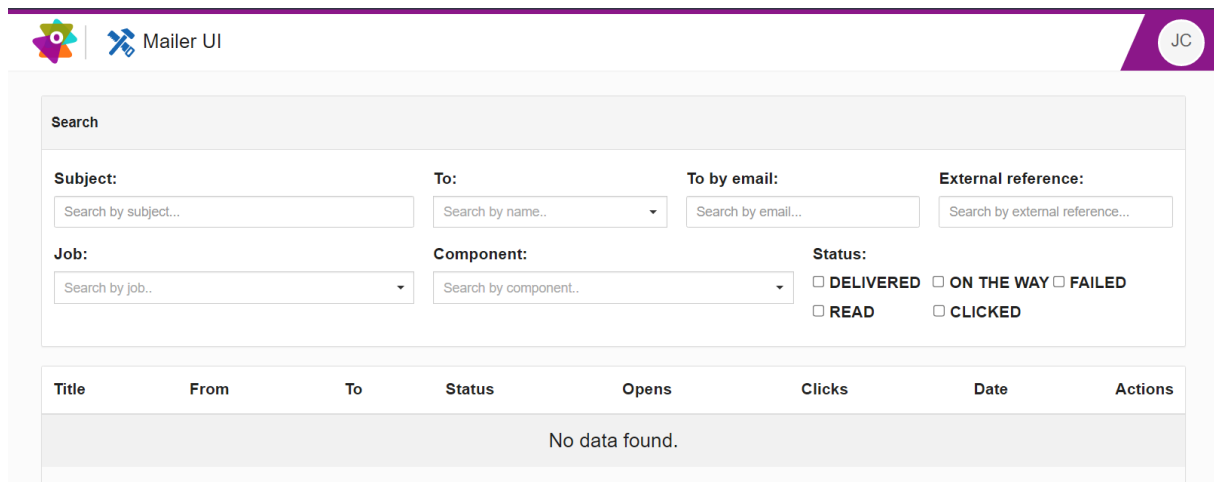
The primary objective of this project is to migrate an existing Angular.js application to React while maintaining visual consistency in the user interface. A crucial aspect of achieving this goal involves replicating the CSS code from the Angular.js application and incorporating it into the React version to ensure a coherent user experience.

Additionally, the project entails the conversion of the Angular.js logic, including the login functionality, to React. This process requires meticulous attention to detail to ensure that the login mechanism in the React implementation mirrors the behavior and functionality of its Angular.js counterpart.

To ensure the comprehensive integration of all functionalities into the React version, it is imperative to adapt and implement the existing logic from the Angular.js application. This involves leveraging React hooks to modify existing API calls and working in tandem with the pre-existing backend endpoints to handle data effectively.

An essential feature of the application revolves around the ability for users to search and filter their emails. It is vital to implement this functionality by utilizing the existing API endpoints for data retrieval and manipulation, ensuring that users can seamlessly access and navigate their emails within the React version.

Overall, the project necessitates significant development efforts to successfully transform the Angular.js application into a fully functional React application, while preserving visual fidelity and incorporating all existing features and functionalities.



The screenshot shows the Mailer UI interface. At the top, there's a header with a logo and 'Mailer UI' text, and a user profile 'JC' on the right. Below the header is a search section with a 'Search' label and a search bar. Underneath are several filter sections: 'Subject:' with a search bar, 'To:' with a dropdown, 'To by email:' with a search bar, 'External reference:' with a search bar, 'Job:' with a dropdown, 'Component:' with a dropdown, and 'Status:' with checkboxes for 'DELIVERED', 'ON THE WAY', 'FAILED', 'READ', and 'CLICKED'. Below these filters is a table with columns: 'Title', 'From', 'To', 'Status', 'Opens', 'Clicks', 'Date', and 'Actions'. The table body is empty, showing 'No data found.'

Figure 9: Angular.js application UI

Title	From	To	Status	Opens	Clicks	Date	Actions
(PRO.-thema) Kwaliteitsinstrumenten : review aangevra...	Redactie	pro.eindredactie@ka...	SENT			Mar 29, 2023 04:40:33	
(PRO.-thema) Kwaliteitsinstrumenten : review aangevra...	Redactie	pro.eindredactie@ka...	SENT			Mar 29, 2023 04:38:31	
(PRO.-thema) Salaris en vergoedingen : review aangevr...	Redactie	pro.eindredactie@ka...	SENT			Mar 29, 2023 02:35:55	
(PRO.-thema) Salaris en vergoedingen : review aangevr...	Redactie	pro.eindredactie@ka...	SENT			Mar 29, 2023 02:34:09	
(PRO.-thema) Salaris en vergoedingen : review aangevr...	Redactie	pro.eindredactie@ka...	SENT			Mar 29, 2023 02:32:11	
(PRO.-thema) Salaris en vergoedingen : review aangevr...	Redactie	pro.eindredactie@ka...	SENT			Mar 29, 2023 02:31:22	
(PRO.-thema) Salaris en vergoedingen : review aangevr...	Redactie	pro.eindredactie@ka...	SENT			Mar 29, 2023 02:23:44	

Figure 10: Angular.js application UI with data

Search

To:

Search by name..

Job:

Search by job..

Component:

Search by component..

Status:

☐ DELIVERED
 ☐ ON THE WAY
 ☐ FAILED
 ☐ READ
 ☐ CLICKED

Title	From	To	Status
(PRO.-thema)...	Redactie	pro.eindredact...	SENT
(PRO.-thema)...	Redactie	pro.eindredact...	SENT

Figure 11: Angular.js application responsive

2.3 Conclusion of the internship assignment

Ticket 1

Set up new React project

Figure 12: ticket 1

The primary task involved the setup of a React application, utilizing a company-provided package. This package facilitated the incorporation of React and Vite, streamlining the setup process. However, certain errors were encountered when attempting to set up the application on a specific computer, while functioning correctly on other machines.

Ticket 2



Figure 13: ticket 2

The task at hand involved the integration of a deployed authentication system to enable user login functionality in the application. The primary objective was to verify the user's authentication status and, if necessary, redirect them to the authentication system. Conversely, if the user was authenticated, access to the application was granted. To facilitate the implementation, the company provided relevant code from another application that had already addressed this requirement. Extensive consultation with team members was undertaken to gain a comprehensive understanding of the codebase and the intricacies of the existing authentication mechanism.

Retrieving and utilizing user data played a crucial role in determining the authorization status and determining the need for redirection to the authentication system. Additionally, the implementation involved creating a header component that displayed the user's initials or profile picture, utilizing the acquired user data. Further functionality included enabling users to click on the profile picture, granting access to settings or allowing them to log out from the application.

To ensure effective data sharing across various components, React Context, a state management tool, was employed. Furthermore, particular attention was paid to replicating the CSS styling of the Angular.js application, aligning the visual presentation of the new project with the existing interface to maintain a consistent user experience.

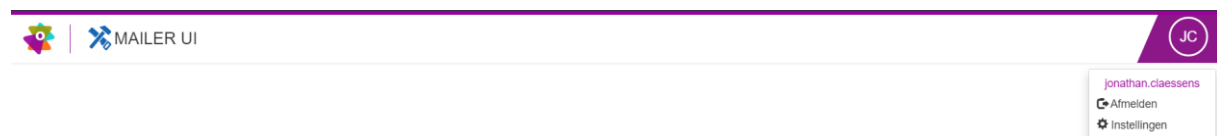


Figure 14: application header

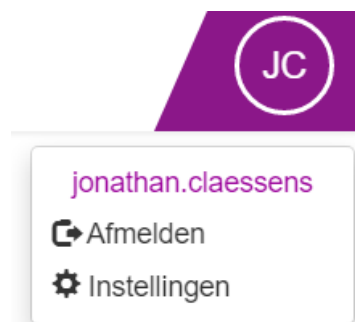


Figure 15: application user dropdown

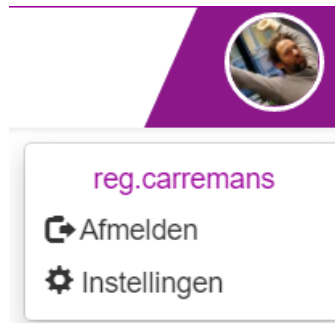


Figure 16: application user with picture dropdown

Ticket 3

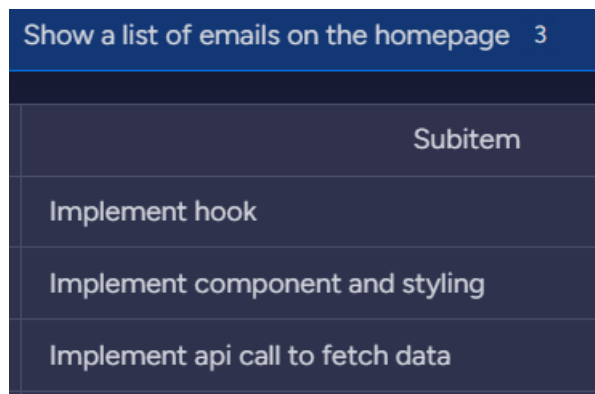


Figure 17: ticket 3

The task involved the implementation of an email display feature for the user, with the aim of replicating the visual appearance of the Angular.js application. To approach this task systematically, it was divided into three subitems. The initial focus was on developing the necessary component and styling, utilizing hardcoded data for testing purposes.

Subsequently, attention shifted towards the backend implementation, specifically the incorporation of a hook to streamline data retrieval. The objective was to enable the retrieval of all email data through a single API call, as opposed to the two API calls required in the Angular.js application.

Lastly, the frontend implementation entailed integrating the API call to fetch the relevant email data and presenting it within the application. To manage the retrieved emails effectively, a state management approach was employed.

Overall, the task involved iterative steps that encompassed component implementation, backend optimization, frontend API integration, and the utilization of state management techniques to ensure a coherent display of user emails.

Title	From	To	Status	Opens	Clicks	Date	Actions
Bevraging welzijn op het werk: WellBe.Vlaanderen Don Bo...	WellBe	frederik.vermeire@ksd...	SENT			May 24, 2023, 14:15:56	
Bevraging welzijn op het werk: WellBe.Vlaanderen	WellBe	greet@gmail.com	SENT			May 24, 2023, 14:15:56	

Figure 18: application mail display

The assigned task entailed the implementation of a customized status feature with associated CSS styling.

(PRO.-thema) Test selector	Redactie	pro.eindredactie@kath...	REJECTED			Apr 13, 2023, 14:19:43	▶
(PRO.-thema) Theme page Test strange characters descripti...	Redactie	pro.eindredactie@kath...	INVALID	1	2	Apr 13, 2023, 11:30:18	▶
(Gedeeld databankitem) ATTACHMENTS Kopie van Leerpla...	Redactie	pro.eindredactie@kath...	SCHEDULED			Apr 11, 2023, 09:41:56	▶
(PRO.-nieuwsbericht) lang nieuwsbericht2	Redactie	pro.eindredactie@kath...	SPAM			Apr 11, 2023, 08:26:13	▶

Figure 19: application status display

The assigned objective involved the implementation of an action button designed to open the email message in a new browser tab.



Figure 20: application action button click result

Ticket 4

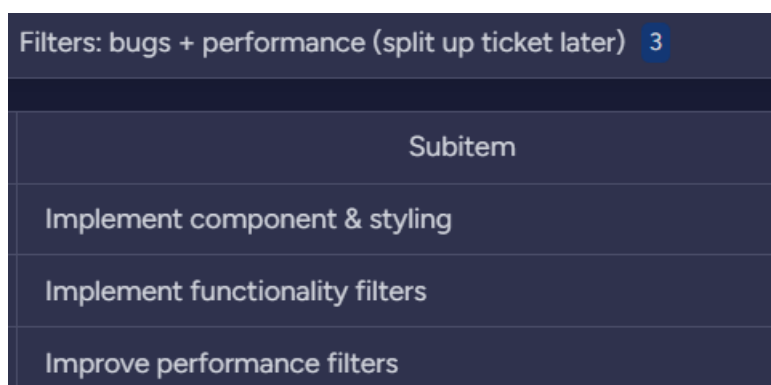


Figure 21: ticket 4

The ticket was subdivided into three distinct subitems to facilitate a structured approach. Initially, the focus was directed towards implementing the component along with its associated styling to ensure visual consistency with the Angular.js application. Subsequently, attention was shifted to the backend, where improvements were made to enhance the efficiency of the filters. This was crucial as certain filters exhibited significant delays in returning the desired results. Following the backend enhancements, the filter functionality was implemented, necessitating adjustments to accommodate discrepancies observed in the Angular.js application.

Efficient communication and coordination were employed to address the inconsistencies in the filters and ensure their proper functioning. Furthermore, meticulous attention was devoted to utilizing the appropriate API calls to maintain consistency and accuracy in the data retrieval process. The outcome of this ticket materialized in the form of a refined and functional implementation.

To facilitate effective management of the filters and real-time updates of the displayed emails, a state management mechanism was employed. This allowed for seamless synchronization between the filter changes and the corresponding updates in the displayed email list.

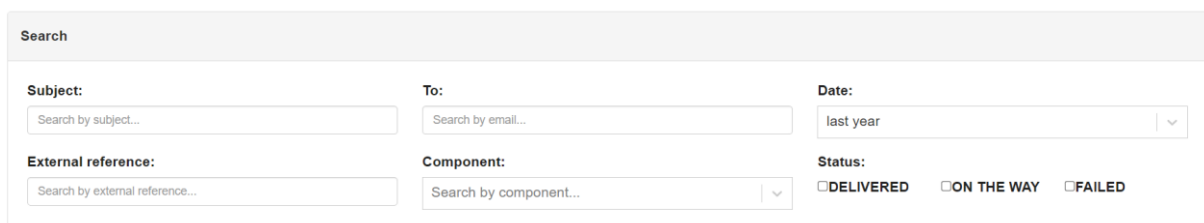
The image shows a search filter interface with a light gray header labeled "Search". Below the header, there are six input fields arranged in two rows of three. The first row contains "Subject:" with a text input "Search by subject...", "To:" with a text input "Search by email...", and "Date:" with a dropdown menu showing "last year". The second row contains "External reference:" with a text input "Search by external reference...", "Component:" with a dropdown menu showing "Search by component...", and "Status:" with three radio buttons labeled "DELIVERED", "ON THE WAY", and "FAILED".

Figure 22: application filter

Furthermore, an additional task involved the implementation of a dropdown functionality. For this purpose, a shared React component provided by the company was utilized, serving as the foundation for both the date and component filters. Notably, in the date filter, a default selection was established to ensure a preselected option.

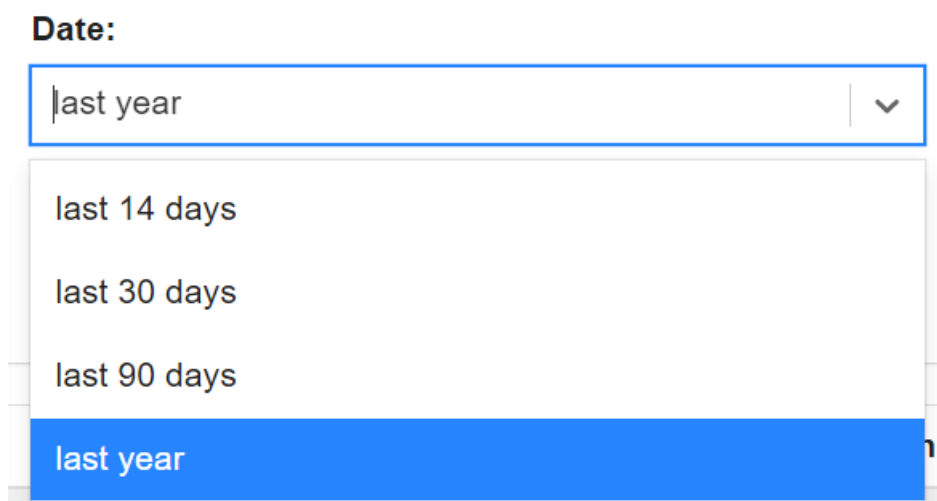
The image shows a dropdown menu for the "Date:" filter. The dropdown is open, showing a list of options: "last 14 days", "last 30 days", "last 90 days", and "last year". The "last year" option is highlighted in blue. The dropdown is styled with a light gray background and a blue border.

Figure 23: application date filter

As part of the filter component implementation, a requirement entailed ensuring that the component was initially unselected upon loading the application.

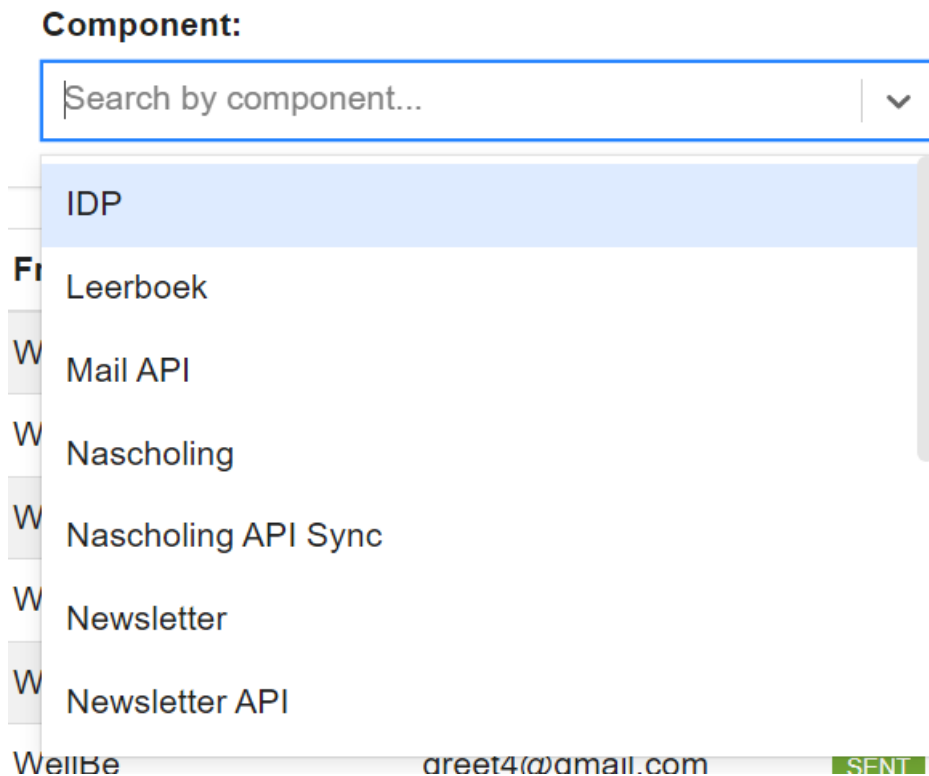


Figure 24: application component filter

Ticket 5

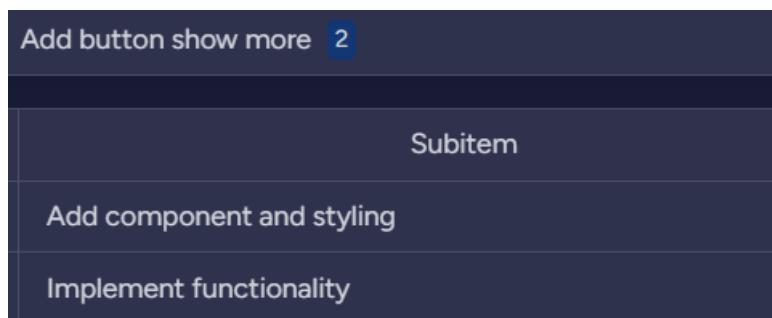


Figure 25: ticket 5

The existing approach of retrieving all mails that matched the applied filters during the data fetch process had potential performance drawbacks, particularly when dealing with a substantial volume of emails. To enhance performance, an implementation strategy was devised to limit the number of retrieved mails. Consequently, a button was introduced to enable users to fetch additional mails selectively, while maintaining a restricted number of retrieved mails at all times for optimal performance.

The implementation process began with the development of the component and associated styling, which consisted of a single button. Subsequently, the functional aspect was addressed by modifying the API call responsible for retrieving the mails. When the user clicked the button, an API call with an increased limit was triggered, allowing for the retrieval of additional mails, which were then updated and displayed in the browser.

To ensure a seamless user experience, the API response included a property indicating whether there were any remaining mails left to be retrieved. Based on this property, the button in the user interface was either shown or hidden accordingly.

By implementing this approach, performance was enhanced by restricting the initial data fetch while providing an option for users to selectively retrieve additional mails as needed, balancing performance considerations with usability.

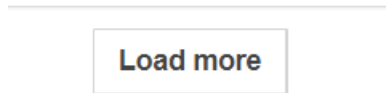


Figure 26: application load more button

Ticket 6

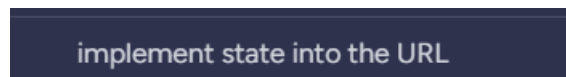


Figure 27: ticket 6

The implementation involved incorporating query parameters into the URL to enable the display and retrieval of filtered emails. When filters were selected, the URL would dynamically update to reflect the applied filter criteria. Consequently, if someone else accessed the same URL, the emails would be retrieved based on the shared filters.

Multiple implementation approaches were available for achieving this functionality. In this case, a tutorial found online demonstrated the utilization of React Router, a widely-used library, to facilitate the integration of query parameters into the URL structure. Furthermore, adjustments were made to ensure that the filters displayed on the application's interface were synchronized with the filters extracted from the URL.

By adopting this implementation strategy, the application allowed for seamless sharing and retrieval of filtered emails through the manipulation of query parameters within the URL.

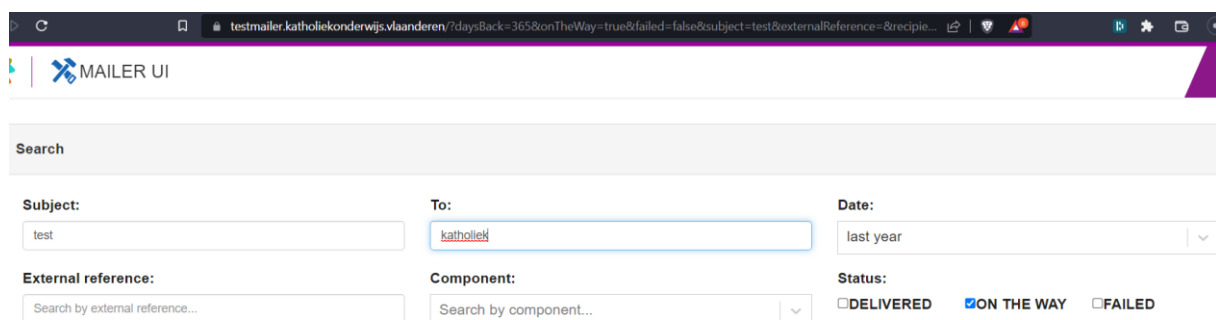


Figure 28: application query parameters

Ticket 7

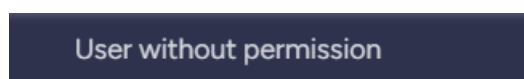


Figure 29: ticket 7

During the testing phase, a bug in the login functionality was identified. Specifically, the application lacked the capability to facilitate user impersonation or login on behalf of other accounts. To address this limitation, a new API call was implemented to evaluate whether a user possessed the necessary privileges to perform the account impersonation action. The response from this API call was then utilized to determine whether the user should be granted or denied access to this feature.

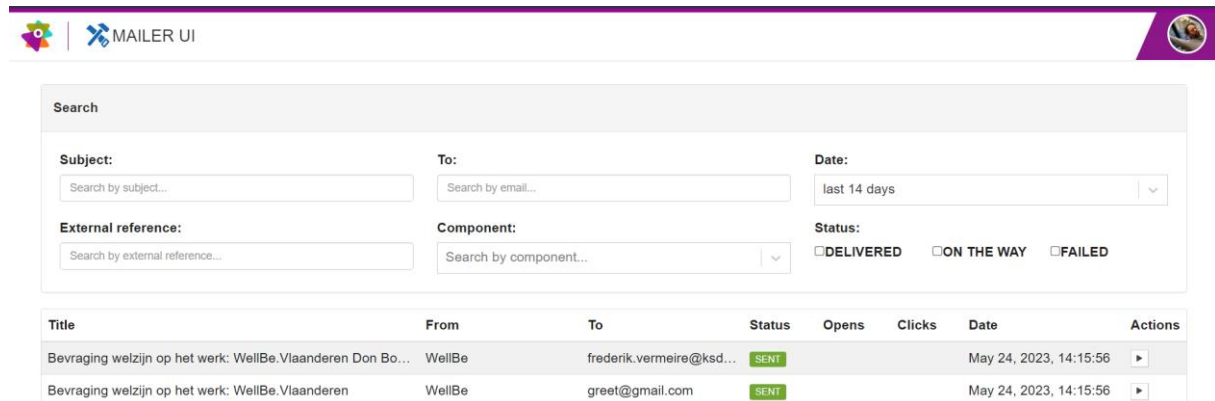


Figure 30: application impersonation success

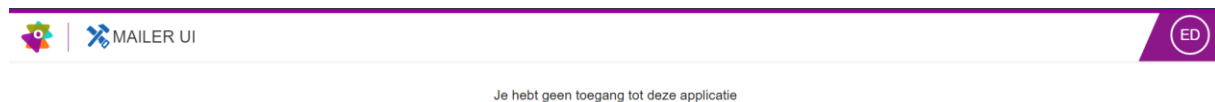


Figure 31: application impersonation failed

Ticket 8

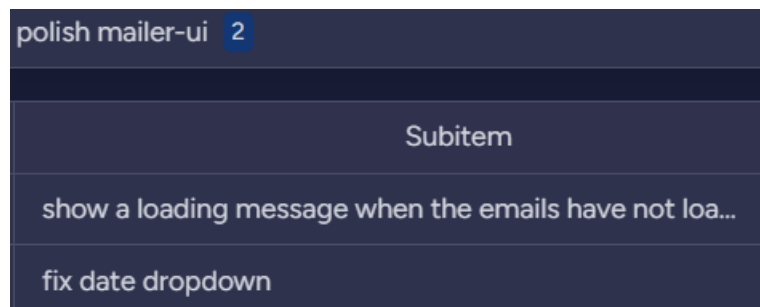


Figure 32: ticket 8

During the process of email retrieval, there was a lack of clarity regarding whether the emails had been successfully fetched or if the operation was still in progress. To address this issue, an implementation was undertaken to display a loading message, providing the user with feedback while waiting for the retrieval process to complete.

Additionally, an issue arose where selecting a value in the dropdown resulted in all values being highlighted as if they were selected. To rectify this behavior, a property was introduced within the shared React component upon its import, allowing for appropriate configuration and resolving the unintended highlighting of all values.

Through these implementations, the user experience was enhanced by providing clear indications of the email retrieval status and rectifying the erroneous highlighting behavior within the dropdown component.

Title	From	To	Status	Opens	Clicks	Date	Actions
Loading ...							

Figure 33: application loading message

Date:

▼

last 14 days

last 30 days

last 90 days

last year

Figure 34: application dropdown css fix

Final result

The following figures depict the visual representation of the completed project.

<div> <div> MAILER UI </div> <div>JC</div> </div>							
<div> <div>Search</div> <div> <div> <div>Subject:</div> <input type="text" value="Search by subject..."/> </div> <div> <div>To:</div> <input type="text" value="Search by email..."/> </div> <div> <div>Date:</div> <input type="text" value="last 14 days"/> ▼ </div> </div> <div> <div> <div>External reference:</div> <input type="text" value="Search by external reference..."/> </div> <div> <div>Component:</div> <input type="text" value="Search by component..."/> ▼ </div> <div> <div>Status:</div> <div> <input type="checkbox"/> DELIVERED <input type="checkbox"/> ON THE WAY <input type="checkbox"/> FAILED </div> </div> </div> </div>							
Title	From	To	Status	Opens	Clicks	Date	Actions
[Nascholing] Inschrijvingsoverzicht voor Wiskunde - Webin...	nascholing@katholiek...	annemiek.hoofman@s...	SENT			Jun 07, 2023, 14:43:37	▶
[Nascholing] Inschrijving voor Wiskunde - Webinar "Wegwij...	nascholing@katholiek...	an.vanhuffel@skynet.be	SENT			Jun 07, 2023, 14:43:36	▶
(PRO.-thema) Decreet internaten : review aangevraagd do...	Redactie	pro.eindredactie@kath...	SENT			Jun 07, 2023, 14:42:55	▶
[Nascholing] Gebruikersnaam Vergeten	nascholing@katholiek...	rosita.cilissen@vti-leu...	SENT			Jun 07, 2023, 14:42:47	▶
(Vacature) Moretus in Ekeren zoekt een algemeen directeur	Redactie	www.eindredactie@ka...	SENT			Jun 07, 2023, 14:40:05	▶
[Nascholing] Inschrijving voor professionalisering internate...	nascholing@katholiek...	lynn.raes@tergroenep...	SENT			Jun 07, 2023, 14:40:04	▶

Figure 35: finished react application

Search

Subject:

Search by subject...

To:

Search by email...

Date:

last 14 days

▼

External reference:

Search by external reference...

Component:

Search by component...

▼

Status:

☐ DELIVERED
☐ ON THE WAY
☐ FAILED

Title	From	To	Status
[Nascholing] In...	nascholing@ka...	annemiek.hoof...	SENT
[Nascholing] In...	nascholing@ka...	an.vanhuffel@...	SENT

Figure 36: finished react application responsive

II. Research topic

1 Research questions, sub-questions, and hypotheses

1.1 Research purpose

The research on designing scalable web applications with Node.js and PostgreSQL is important for several reasons. Firstly, as the demand for web applications continues to grow, it is essential to ensure that these applications can handle the increased traffic and user load without crashing or slowing down. This requires a deep understanding of the best practices for caching and load balancing, which can help to optimize performance and ensure that the application remains scalable over time.

Secondly, Node.js and PostgreSQL are popular technologies for building web applications, and there is a need for more research on how to effectively use these technologies for scalable applications. This research can help developers to make informed decisions when designing and building web applications and can contribute to the overall development of best practices in the field.

Finally, the research on caching and load balancing can have significant implications for businesses that rely on web applications. A poorly designed and optimized application can result in lost revenue, frustrated users, and damage to the reputation of the company. By understanding and implementing the best practices for caching and load balancing, businesses can ensure that their web applications are reliable, efficient, and scalable, providing a better experience for users and contributing to the overall success of the business.

1.2 The main research question is:

What are the best practices for designing scalable web applications using Node.js and PostgreSQL, specifically in regard to caching and load balancing?

1.3 The sub-questions that need to be answered are:

- What is Node.js?
- What is PostgreSQL?
- What is caching and its importance for web scalability?
- Why is load balancing important for web scalability?
- Best practices for caching in Node.js and PostgreSQL web applications?
- Real-world examples of successful Node.js and PostgreSQL web applications with caching and load balancing?
- Challenges and trade-offs in implementing caching and load balancing in Node.js and PostgreSQL web applications?

Hypothesis

Designing scalable web applications is crucial to ensure that users can access and interact with the application smoothly, even during periods of high traffic or heavy data processing. In the case of Node.js and PostgreSQL, effective caching and load balancing strategies can help to reduce server load and improve response times, ultimately leading to a better user experience.

By implementing caching mechanisms, such as in-memory caching or content delivery networks, the application can optimize data retrieval and transmission, reducing the burden on the server and improving response times. Similarly, load balancing techniques can distribute incoming requests across multiple servers, preventing overloading and ensuring reliable application performance.

By leveraging these best practices, web applications can be designed to scale efficiently and meet user demands, even as traffic and data processing requirements increase over time.

2 Research method

The research for this paper is focused on investigating the best practices for designing scalable web applications using Node.js and PostgreSQL, with a particular focus on caching and load balancing. To test the effectiveness of these practices, we develop a test application and subject it to high traffic simulations using load testing tools.

However, before beginning this experimental phase, we conduct a thorough review of existing literature and expert opinions to identify the most promising strategies and techniques. This involves answering fundamental questions about web application design and identifying the specific use cases and requirements that are most relevant to the project.

By starting with a strong theoretical foundation and clear understanding of the project goals, the resulting research provides practical insights and recommendations for improving web application scalability using Node.js and PostgreSQL.

3 Research elaboration

3.1 Literature review

3.1.1 What is Node.js?

Node.js

Node.js is an event-driven JavaScript runtime that supports asynchronous programming. It is specifically designed for developing scalable network applications. Whenever a connection is established, a corresponding callback function is executed. If there is no work to be processed, Node.js will enter a sleep state, leading to optimal resource utilization. [1]

```
const http = require('http');

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

Figure 37: Node.js code example

In comparison to the prevalent concurrency model utilizing OS threads, Node.js presents an alternative approach that is characterized by enhanced efficiency and user-friendliness. The adoption of thread-based networking, a commonly employed method, is associated with inefficiencies and difficulties in management. Conversely, Node.js circumvents concerns regarding process dead-locking by operating without locks. [1]

In contrast to conventional approaches, Node.js reduces reliance on direct input/output (I/O) operations, resulting in unimpeded process flow unless synchronous methods from the Node.js standard library are employed. This non-blocking nature facilitates the development of scalable systems effortlessly within the Node.js environment. [1]

Thread-based networking is a networking model in which each client connection is handled by a separate thread. In this model, a server creates a new thread for each client that connects to it, allowing the server to handle multiple clients simultaneously. Each thread is responsible for handling the communication with a single client, and can operate independently of the other threads. [2]

Event loop

The event loop in Node.js enables the execution of non-blocking I/O operations, even though JavaScript itself is single-threaded. This is achieved by delegating operations to the system kernel whenever possible. [3]

Modern kernels, which are typically multi-threaded, can handle multiple operations concurrently. When a particular operation is completed, the kernel notifies Node.js, allowing the corresponding callback to be added to the poll queue for eventual execution. The event loop manages this process, ensuring efficient handling of events and callbacks. [3]

The event loop in Node.js follows a specific order of operations. It begins with the execution of timers, followed by pending callbacks, idle and prepare operations, and the poll phase. The poll phase handles incoming connections, data, and other events. After the poll phase, the check phase is executed, and finally, the close callbacks phase. [3]

Each phase of the event loop maintains a First-In-First-Out (FIFO) queue of callbacks to be executed. Upon entering a phase, specific operations related to that phase are performed, and then the callbacks in the queue are executed until the queue is emptied or the maximum number of callbacks is reached. Once this occurs, the event loop transitions to the next phase and repeats the process. [3]

Notably, operations within each phase can schedule additional operations, and new events processed during the poll phase are enqueued by the kernel. Consequently, the poll phase can continue running for an extended period if there are long-running callbacks. This behavior is further elucidated in the sections on timers and the poll phase, providing more detailed insights into the event loop's behavior and dynamics. [3]

Overview of blocking vs non-blocking

In the Node.js environment, the term "blocking" pertains to the situation where the execution of additional JavaScript code is halted until the completion of a non-JavaScript operation. This occurrence arises due to the event loop's inability to continue running JavaScript while a blocking operation is ongoing. [4]

However, it is important to note that in the specific context of Node.js, JavaScript code exhibiting poor performance as a result of CPU-intensive tasks, rather than waiting for non-JavaScript operations like I/O, is typically not categorized as blocking. The predominant blocking operations in Node.js involve the utilization of synchronous methods from the Node.js standard library that leverage the libuv library. Furthermore, it should be acknowledged that native modules may also incorporate blocking methods. [4]

To mitigate this issue, the Node.js standard library offers asynchronous versions of all I/O methods, which are designed to be non-blocking and accept callback functions. Moreover, certain methods also provide blocking counterparts tailored for specific use cases. [4]

Comparing code

To exemplify the concept of synchronous and asynchronous methods, the following code examples are presented. Blocking methods execute synchronously and non-blocking methods execute asynchronously. Figure 38 demonstrates a synchronous file read using the File System module, while Figure 39 shows an equivalent asynchronous example. [4]

```
const fs = require("fs");
const data = fs.readFileSync("/file.md"); // blocks here until file is read
```

Figure 38: synchronous file read example

```
const fs = require("fs");
fs.readFile("/file.md", (err, data) => {
  if (err) throw err;
});
```

Figure 39: asynchronous file read example

The example in Figure 38 may seem simpler than the one in Figure 39, but it has the disadvantage of blocking the execution of any additional JavaScript until the entire file is read. [4]

To provide a more detailed example of synchronous file reading, Figure 40 expands on the code in Figure 38. Figure 41, on the other hand, demonstrates an asynchronous file reading example similar to Figure 39. [4]

```
const fs = require("fs");
const data = fs.readFileSync("/file.md"); // blocks here until file is read
console.log(data);
moreWork(); // will run after console.log
```

Figure 40: Synchronous file read example expanded

```
const fs = require("fs");
fs.readFile("/file.md", (err, data) => {
  if (err) throw err;
  console.log(data);
});
moreWork(); // will run before console.log
```

Figure 41: Asynchronous file read example expanded

In Figure 40, the `console.log()` function will be called before the `moreWork()` function, as the file read operation is blocking the execution of any additional JavaScript until it is completed. This can have a negative impact on the application's performance and throughput. [4]

In Figure 41, on the other hand, the `fs.readFile()` method is non-blocking, meaning that JavaScript execution can continue and the `moreWork()` function will be called first. This design choice enables higher throughput and better performance, as the application can continue to execute other tasks while the file read operation is being performed asynchronously. [4]

Node.js is an open-source, cross-platform runtime environment designed for the development of server-side and networking applications that prioritize speed and scalability. It operates on the V8 JavaScript runtime engine and leverages an event-driven, non-blocking I/O architecture, rendering it highly efficient and well-suited for real-time application scenarios. [5]

Node.js architecture summary

Node.js architecture is designed to handle asynchronous tasks in an event-driven, single-threaded environment. Node.js uses a non-blocking I/O model, which allows the main thread to switch between tasks while waiting for the completion of an asynchronous request. When a request

containing both synchronous and asynchronous tasks is received, the main thread executes the synchronous portion and delegates the asynchronous part to a background thread. [6]

Although there may be multiple background threads running, Node.js is still considered to be single-threaded because all requests are received and managed by a single event loop. Once the background thread completes the asynchronous task, it notifies the event loop through a callback function, allowing the main thread to resume processing the request. [6]

Overall, Node.js architecture is optimized for handling I/O-bound tasks, providing high performance and scalability. By using an event-driven, non-blocking I/O model, Node.js ensures that the main thread is always busy and can handle multiple requests concurrently, without waiting for expensive I/O operations to complete. [6]

3.1.2 What is PostgreSQL?

PostgreSQL

PostgreSQL is a stable open-source database system that supports various functions of SQL, including foreign keys, subqueries, triggers, and user-defined types and functions. It is designed to store data for a wide range of applications, such as mobile, web, geospatial, and analytics. It also offers features that help to scale and manage data workloads effectively. [7]

PostgreSQL is a powerful and versatile open-source database system that offers a wide range of features and capabilities that are comparable to commercial database management systems. This makes it an appealing choice for businesses of all sizes. [8]

Distinctive features and advantages

Object-relational database

PostgreSQL is classified as an object-relational database management system (ORDBMS) that incorporates both the conventional relational model and additional functionalities typically found in object databases. By encompassing features like table inheritance and function overloading, PostgreSQL exhibits remarkable flexibility and resilience, setting it apart from other database systems. [9]

Concurrency

PostgreSQL exhibits remarkable concurrency capabilities, enabling efficient multitasking without the need for read locks. This is achieved through the implementation of MVCC, a mechanism that guarantees the ACID properties of transactions. By adhering to ACID compliance, PostgreSQL demonstrates its scalability by effectively accommodating a substantial volume of concurrent users and vast amounts of data. [9]

SQL and NoSQL

PostgreSQL serves as a versatile database management system, supporting both traditional SQL-based relational data storage and functioning as a NoSQL solution for storing JSON documents. This adaptability offers potential cost savings and enhanced security capabilities. By utilizing a single database management system, organizations can streamline operations by eliminating the need to engage additional expertise for configuring, administering, securing, and updating multiple database solutions. This consolidation of functionalities contributes to operational efficiency and resource optimization. [10]

Data integrity

PostgreSQL exhibits a strong commitment to safeguarding data integrity, particularly at the transactional level, thereby mitigating the risk of data corruption. Notably, PostgreSQL incorporates essential features such as foreign keys, joins, views, triggers, and stored procedures, which enhance its data management capabilities across various programming languages. Furthermore, PostgreSQL offers three distinct levels of transaction isolation, namely Read Committed, Repeatable Read, and Serializable, providing users with flexible options for balancing consistency and concurrency in their applications. These robust features contribute to PostgreSQL's reputation as a reliable and versatile database management system. [11]

Extensibility

PostgreSQL demonstrates exceptional versatility through its comprehensive range of extensions, effectively catering to diverse use cases. Its extensibility enables users to address specific requirements, encompassing support for specialized data types and advanced logging capabilities. Furthermore, PostgreSQL offers the flexibility for users to develop their own extensions or engage with PostgreSQL vendors to tailor the system to their unique needs. This adaptability underscores PostgreSQL's capability to accommodate a wide spectrum of use cases, ensuring a highly customizable and tailored database solution. [10]

Community support

PostgreSQL exemplifies a robust and dependable database system that has garnered a formidable reputation over the course of more than two decades. As an established entity within the realm of Open Source databases, PostgreSQL competes with industry giants like Oracle, Sybase, and IBM, underscoring its significance in the field. With a steadfast commitment to professional maintenance and development, PostgreSQL offers a reliable platform for running intricate, data-intensive applications. The advantages associated with PostgreSQL are substantial, particularly in the realm of facilitating informed decision-making by delivering tangible and actionable insights. Investing in a database system of this caliber ensures the acquisition of meaningful information critical to organizational success. [12]

Code quality

PostgreSQL adopts a rigorous quality assurance process where each line of code undergoes comprehensive scrutiny by multiple domain experts. Emphasizing a community-driven development approach, PostgreSQL benefits from an efficient bug reporting system, swift issue resolution, and prompt verification of fixes. Consequently, PostgreSQL emerges as a highly reliable and stable database system, fortified by a robust framework that promotes meticulous code evaluation and continual improvement. [10]

PostgreSQL is a highly reliable, scalable, stable, and secure open-source database management system that is free to use and modify. It offers compatibility with various platforms using all major languages and middleware, supports complex data types, and has excellent implementation of core relational features. PostgreSQL is a great choice for scientific data collection works and is known for protecting data integrity at the transaction level, making it less vulnerable to data corruption.

3.1.3 What is caching and its importance for web scalability?

What is caching?

According to OAuth, caching is "a mechanism to improve the performance of any type of application" by storing and accessing data from a cache, which is a software or hardware component designed to serve future requests for the same data faster. Caching allows for efficient reuse of previously computed or retrieved data, with the requested data being searched in the cache first upon arrival of a new request. A cache hit occurs when the requested data is found in the cache, while a cache miss occurs when it cannot be found. Retrieving the required data from a cache is faster than recomputing the result or reading it from the original data store, making caching an effective tool for enhancing system performance. As such, the more requests that can be served from a cache, the faster the system will operate. [13]

Why is caching important?

Caching is a crucial aspect of computer technology that allows developers to achieve significant performance improvements. This is essential because neither users nor developers want applications to take a long time to process requests. Waiting for loading messages can be frustrating for users and can impact the reputation of the application. Therefore, offering high performance has become an unavoidable concept in the technology world, and caching has become omnipresent. As a result, proper implementation of caching systems is essential to compete with the multitude of applications on the market. [13]

Caching also has other benefits such as reducing overhead, including network overhead, and minimizing CPU usage, especially for complex requests. This can prolong the life of machines or servers and reduce infrastructure costs. Avoiding making new requests or reprocessing data can significantly decrease the overall number of requests needed, thus reducing the cost of network communication between services. These side effects make caching a highly valuable tool for developers and businesses alike. [13]

In networking, overhead refers to the processing time and transmitting time required for the TCP/IP protocol. [14]

Types of caching

In-memory caching

In-memory caching is an approach where data is stored directly in RAM, leveraging its assumed higher speed compared to traditional storage systems. Key-value databases are commonly used for implementing in-memory caching. These databases consist of sets of key-value pairs, where each piece of data is identified by a unique key and associated with a corresponding value representing the cached data. By specifying the key, developers can retrieve the desired data from the key-value database efficiently. This caching solution offers speed, efficiency, and simplicity, making it a preferred choice for developers seeking to build a caching layer. [13]

Database caching

Database caching is another common approach to improving performance in applications that rely heavily on databases. By caching query results, a database can avoid executing the same query repeatedly, and instead provide the previously cached data immediately. This can significantly reduce the overhead associated with executing queries and accessing disk-based storage. [9]

Most databases come with built-in caching mechanisms, which are typically implemented using a hash table that stores key-value pairs. The key is used to identify the query, while the value is the result set returned by the query. When a new query is executed, the database checks its cache to see

if the same query has been executed before. If it has, the database retrieves the cached result set and returns it to the application. This approach can provide significant performance gains, especially for read-heavy workloads. [9]

Web caching

Web caching is a technique used to improve the performance of web pages by storing frequently accessed resources for quick access. There are two main subcategories of web caching: web client caching and web server caching. Web client caching is implemented on the user's machine, typically as part of the web browser. It stores the page resources, such as text, images, stylesheets, scripts, and media files on the client side. The next time the same page is accessed, the browser can retrieve the previously cached resources, avoiding the need to download them from the network. Web server caching, on the other hand, stores resources server-side for reuse. It is especially useful for dynamically generated content that takes time to create, as it reduces the workload on the server and improves the page delivery speed. However, it is not recommended for static content. [9]

CDN caching

CDN caching is a technology that stores content, such as web pages, stylesheets, scripts, and media files, in proxy servers. These servers act as gateways between the user and the origin server, caching resources for faster access. When a user requests a resource, the proxy server checks if it has a copy in its cache. If so, the resource is immediately delivered to the user, otherwise, the request is forwarded to the origin server. Since the proxy servers are located in numerous locations worldwide, user requests are dynamically routed to the nearest one, reducing network latency and the number of requests made to origin servers. This makes CDN caching a valuable technology for improving website performance and reducing network costs. [13]

3.1.4 Why is load balancing important for web scalability?

What is load balancing?

Load balancing is the process of distributing network traffic across multiple backend servers to improve efficiency, reliability, and scalability of high-traffic websites. As modern websites must handle large volumes of concurrent requests, load balancers act as the "traffic cop" by directing incoming client requests to available servers, optimizing speed and capacity utilization. Load balancers also ensure that no single server is overworked and redirects traffic to remaining servers if one goes down. This provides high availability and flexibility to add or remove servers as demand dictates. Overall, load balancing is a critical component of modern computing best practices to optimize server utilization and enhance website performance. [15]

Here's a visual explanation of load balancing: "In this manner, a load balancer performs the following functions:

- Distributes client requests or network load efficiently across multiple servers
- Ensures high availability and reliability by sending requests only to servers that are online
- Provides the flexibility to add or subtract servers as demand dictates

" [15]

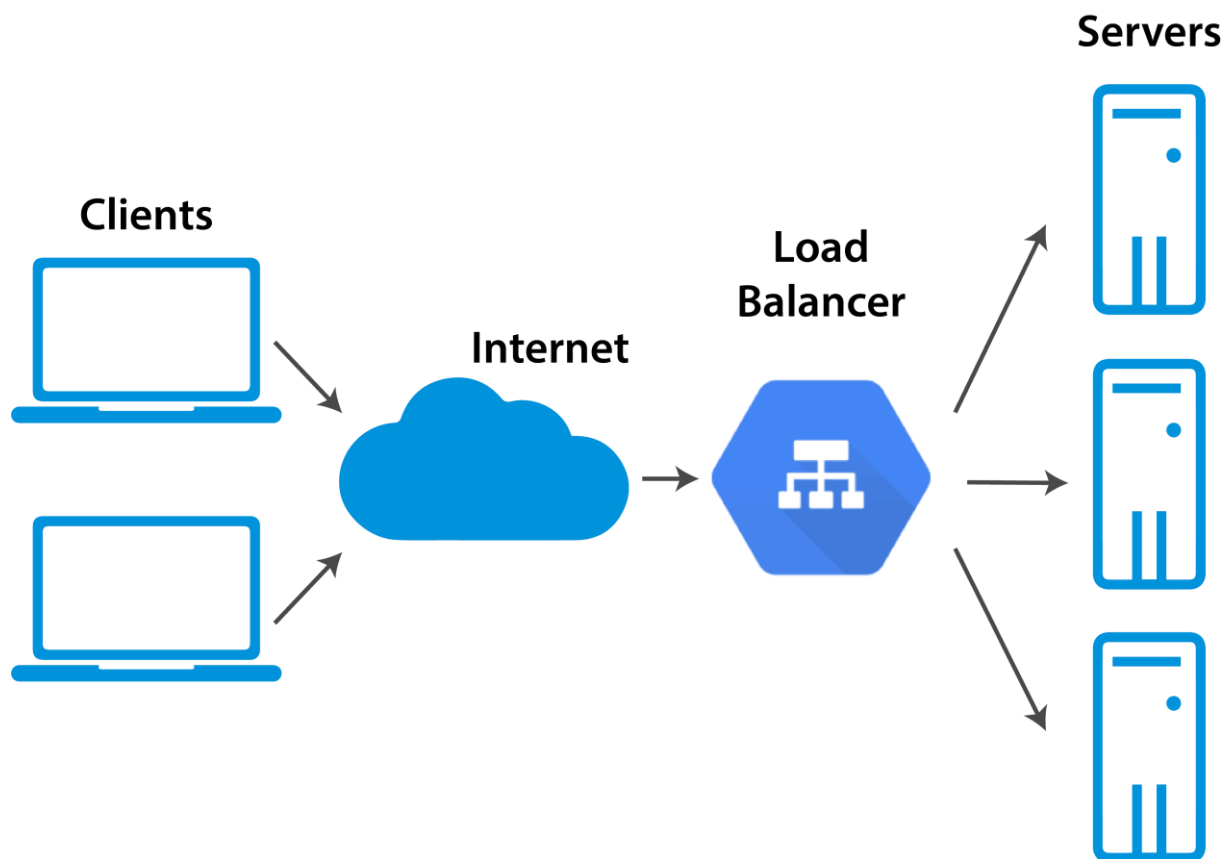


Figure 42: load balancing diagram

Benefits of load balancing

Load balancers offer several benefits to website owners, including improved security, performance, resilience, and scalability. In terms of security, load balancers can add additional layers of protection to websites without any changes to the application. They can protect against emerging threats, authenticate user access, protect against DDoS attacks, and simplify PCI compliance for websites that process credit cards. [16]

In terms of performance, load balancers can reduce the load on web servers, optimize traffic for a better user experience, and support HTTP/2. They can also enable SSL offloading, traffic compression, and traffic caching, resulting in faster and more efficient websites. [16]

HTTP/2 introduces several features that improve the efficiency of HTTP over the network. For instance, it allows servers and clients to choose between HTTP 1.1, 2.0, or other non-HTTP protocols. It also improves page load speed by using data compression of HTTP headers, pipelining of requests, and multiplexing of requests over the same TCP connection. HTTP/2 works with both HTTP and HTTPS-based URIs and makes encryption a standard. [17]

In the context of HTTP/2, multiplexing refers to the ability to send multiple requests and responses simultaneously over a single TCP connection. This improves the efficiency of HTTP/2 by reducing latency and increasing the use of network resources. Multiplexing allows multiple requests to be in flight at the same time, which can significantly improve the performance of a website. [18]

SSL offloading is the process of removing the SSL-based encryption from incoming traffic to relieve a web server of the processing burden of decrypting and/or encrypting traffic sent via SSL. [19]

Regarding resilience, load balancers can accommodate failed and under-performing components to maintain user service. They can provide continued service if a web server fails, workaround busy servers, create a highly available site, and simplify business continuity. [16]

One of the key benefits of using a load balancer is its ability to facilitate scalability without any disruption to user service. The load balancer simplifies the process of adding additional servers or adjusting cloud hosting to meet varying demand, and can seamlessly redirect users to an alternate site in the event of a site outage. This also helps avoid costly and disruptive upgrades to existing servers. In short, a load balancer can help ensure business continuity and optimize website performance. [16]

3.1.5 Best practices for caching in Node.js and PostgreSQL web applications?

Caching

It is essential to take into account three key factors before selecting a caching solution: “

1. The type of data being cached.
2. How the data is read and written (the data access strategy).
3. How the cache evicts old or outdated data (the eviction policy).

” [20]

Data access strategies

The data access pattern, also referred to as the data access strategy, determines the relationship between the data source and the caching layer. It is a crucial factor to consider when implementing a caching solution, as it can significantly impact the effectiveness of the cache. In this section, we will explore common data access patterns and their advantages and disadvantages. [20]

Cache-Aside

The Cache-Aside strategy is a commonly used caching approach where a cache sits on the side and the application communicates directly with both the cache and the database. The cache and the primary database are not connected, and all operations are managed by the application. This approach is illustrated in the figure presented below. [21]

Cache-Aside

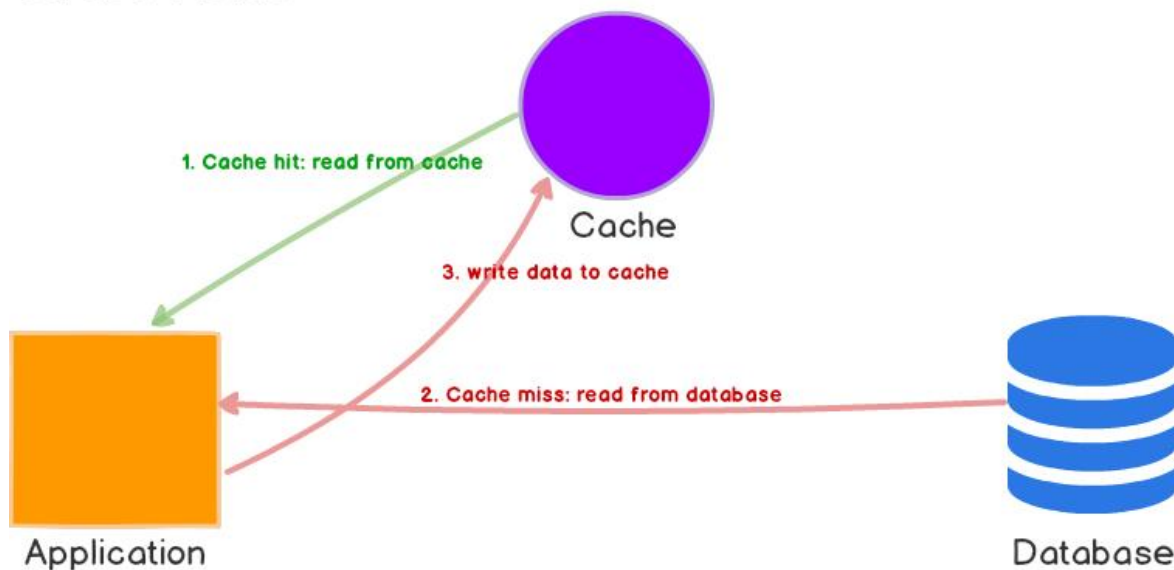


Figure 43: cache-aside concept

This is an explanation of the cache-aside approach and how it works: “

1. The application first checks the cache.
2. If the data is found in cache, we’ve cache hit. The data is read and returned to the client.
3. If the data is not found in cache, we’ve cache miss. The application has to do some extra work. It queries the database to read the data, returns it to the client and stores the data in cache so the subsequent reads for the same data results in a cache hit.

” [21]

Use Cases, Pros and Cons

The cache-aside caching approach is commonly used in read-heavy workloads and is implemented using general-purpose caching tools such as Memcached and Redis. Cache-aside caching provides resilience to cache failures by allowing the system to operate even when the cache cluster is down, though performance may suffer. [21]

Additionally, data models in the cache can differ from those in the database, and data generated from multiple queries can be stored against specific request IDs. [21]

The most common write strategy in cache-aside caching is to write data directly to the database, which can lead to inconsistency between the cache and the database. Developers can use TTL to continue serving stale data until it expires, or they can invalidate the cache entry or use a suitable write strategy to guarantee data freshness. [21]

Read-Through

Read-Through

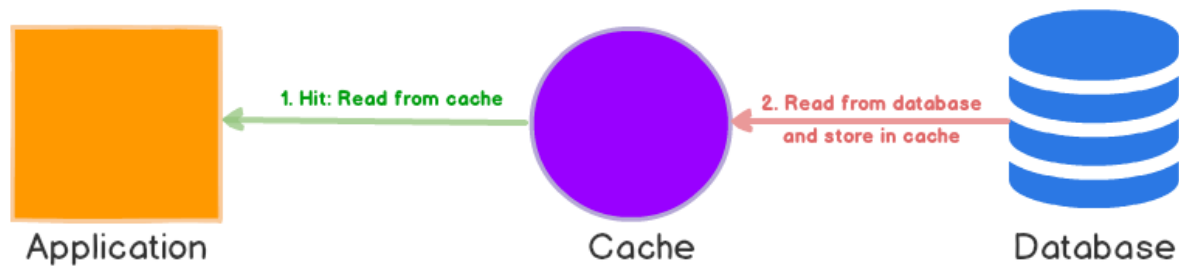


Figure 44: read-through concept

According to the source, read-through cache is a caching approach that is positioned in-line with the database. In the event of a cache miss, this approach loads the missing data from the database, adds it to the cache, and sends it back to the application. Similar to cache-aside strategy, read-through cache also uses a lazy loading approach, meaning that data is only loaded when it is first requested. [21]

Use Cases, Pros and Cons

While these strategies are similar, there are some key differences between them. In cache-aside, the application is responsible for fetching data from the database and populating the cache, whereas in read-through, this logic is usually handled by a library or standalone cache provider. Additionally, the data model in read-through cache must be the same as that of the database, whereas in cache-aside, it can differ. [21]

Read-through caches are most effective for read-heavy workloads where the same data is requested repeatedly, but they can incur a penalty when data is first requested and loaded into the cache. To mitigate this, developers often manually "warm" or "pre-heat" the cache with commonly requested data. However, inconsistencies between cache and database data are still possible and must be addressed with a suitable write strategy. [21]

Write-Through

The source describes a write strategy for caching data in which the data is written to the cache first and then to the main database. The cache is in-line with the database and all writes go through the cache to ensure consistency. The use of this write strategy helps in maintaining consistency between the cache and the main database. [21]

Write-Through

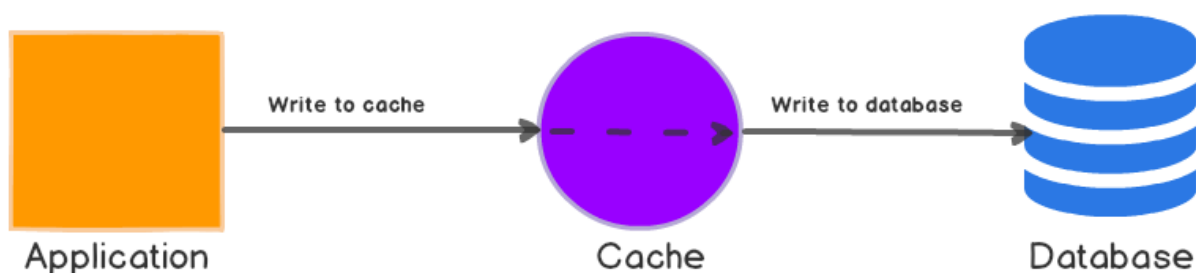


Figure 45: write-through concept

The following happens whenever the application writes or updates a value: “

1. The application writes the data directly to the cache.
2. The cache updates the data in the main database. When the write is complete, both the cache and the database have the same value and the cache always remains consistent.

” [21]

Use Cases, Pros and Cons

Write-through caches can introduce additional write latency as data is written to the cache first and then to the database, but when paired with read-through caches, they can provide data consistency guarantees without the need for cache invalidation. This is possible if all writes to the database go through the cache, making it a useful approach in certain use cases. [21]

Write-Around

The source describes this as the following: “Here, data is written directly to the database and only the data that is read makes it way into the cache.” [21]

Use Cases, Pros and Cons

The write-around caching strategy can be effectively combined with read-through and cache-aside approaches to achieve good performance in scenarios where data is infrequently read or never read. This pattern is suitable for real-time logs, chatroom messages, and other similar use cases. [21]

Write-Back or Write-Behind

Write-Back

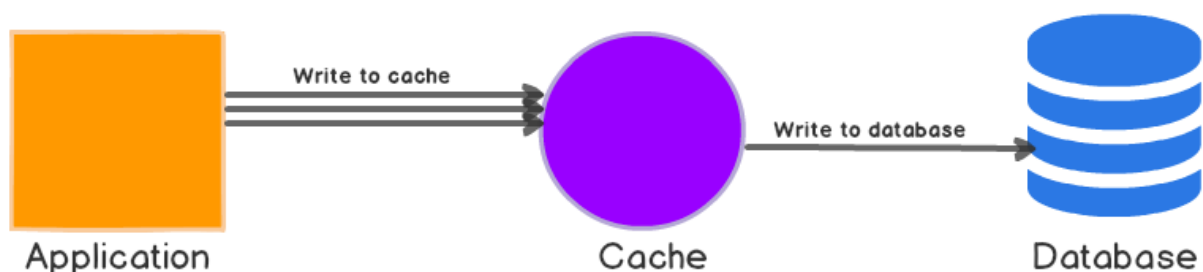


Figure 46: write-back concept

This source explains the concept of Write-Back caching, where an application writes data to the cache and receives an acknowledgement immediately, but the data is written back to the database asynchronously at a later time. [21]

This approach is similar to Write-Through caching, but with a crucial difference in how data is updated in the main database. While Write-Through synchronously updates data in the main database, Write-Back does so asynchronously, resulting in faster write operations from the application's perspective since only the cache needs to be updated before returning a response. This mechanism is also referred to as write-behind. [21]

Use Cases, Pros and Cons

The source discusses the benefits of using write-back caches, particularly for write-heavy workloads. When combined with read-through, write-back caches are suitable for mixed workloads, where the most recently updated and accessed data is always available in the cache. [21]

The use of write-back caches reduces overall writes to the database, which can lead to a reduction in costs if the database provider charges by the number of requests. Some developers use Redis for both cache-aside and write-back to better absorb spikes during peak loads, but this comes with the risk of data loss in the event of cache failure. [21]

Finally, the source points out that most relational database storage engines have write-back caches enabled by default in their internals, where queries are first written to memory and eventually flushed to the disk. [21]

Refresh-ahead pattern

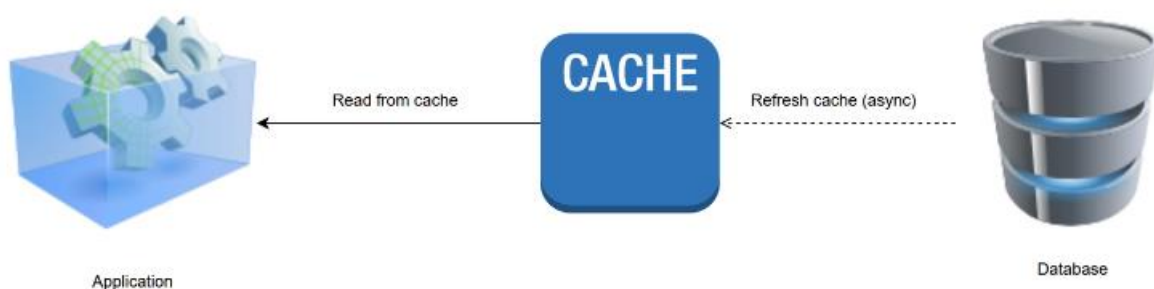


Figure 47: refresh-ahead concept

The refresh-ahead caching pattern involves proactively refreshing cached data before it expires. This is done asynchronously to avoid any performance slowdowns during the retrieval of expired objects from the data store. This pattern ensures that frequently accessed data is always available in the cache and can significantly improve application performance. [20]

Use Cases, Pros and Cons

The refresh-ahead caching strategy is beneficial when reading data from the data store is costly. In this strategy, frequently accessed cached data is refreshed before they expire, keeping the frequently accessed cache entries in sync. The refresh-ahead caching strategy is ideal for latency-sensitive workloads such as live sports scoring sites and stock market financial dashboards. However, the cache needs to accurately predict which cache items are likely to be needed in the future to avoid incurring unnecessary database reads. Otherwise, inaccurate predictions can cause unnecessary database reads and make this strategy less effective. [20]

Summary

Various caching strategies have been discussed, along with their advantages and limitations. It is important to carefully evaluate goals and data access patterns before selecting a caching strategy to avoid additional latency and reduced benefits. It is recommended to understand read/write patterns and choose the best strategy or combination of strategies to optimize system performance. Selecting the appropriate caching strategy is particularly important in high-throughput systems where memory and server costs are a concern. [21]

Cache Eviction Policy

The size of a cache is usually limited compared to the size of the database, so it is necessary to store only the items that are needed and remove redundant entries. A cache eviction policy ensures that the cache does not exceed its maximum limit by removing older objects from the cache as new ones are added. There are several eviction algorithms to choose from, and the best one will depend upon the needs of your application. [20]

When choosing an eviction policy, keep in mind that it isn't always appropriate to apply a global policy to every item in the cache. If a cached object is very expensive to retrieve from the data store, it may be beneficial to retain this item in the cache, regardless of whether it meets the requirements for eviction. A combination of eviction policies may also be required to achieve the optimal solution for your use case. In this section, we'll take a look at some of the most popular algorithms used in production environments. [20]

Least Recently Used (LRU)

The LRU policy is commonly used in cache eviction algorithms. It is based on organizing cache items according to the order of use, so that the most recently used items are placed at the top of the cache and the least recently used ones at the bottom. [20]

The LRU algorithm updates the timestamp of an object every time it is accessed and moves it to the top of the cache. When the cache reaches its maximum capacity, the algorithm evicts items at the bottom of the cache based on their last access time. This makes it easy to identify which items should be evicted when it's time to clean up the cache. [20]

Least Frequently Used (LFU)

The LFU cache eviction algorithm tracks how frequently cached items are accessed and evicts items with the lowest usage count. The algorithm increments a counter on a cached object every time it is accessed, and items with the lowest count are evicted when it's time to free up space. [20]

LFU is particularly useful in scenarios where the access patterns of the cached objects remain relatively constant, such as in CDN asset caching, where frequently accessed objects are retained and rarely accessed objects are removed. This algorithm also helps to evict items that experience a spike in usage at a particular time, but whose access frequency drops significantly thereafter. [20]

Most Recently Used (MRU)

The MRU eviction policy is an algorithm used to analyze cache items based on the recency of their last access. Unlike the LRU algorithm, MRU discards the most recently used objects from the cache instead of the least recently used ones. [20]

This policy is ideal when it is unlikely that a recently accessed object will be used again soon, and therefore it can be removed from the cache immediately. A good example of this is removing booked flight seats from the cache immediately after booking, as they are no longer relevant for a subsequent booking application. [20]

First In, First Out (FIFO)

FIFO is a cache eviction policy that removes the oldest cached items in the order they were added, regardless of how frequently or how many times they were accessed. [20]

Cache expiration

Cache expiration is an important aspect of a cache policy that determines how long a cached item is retained. The expiration policy is typically assigned to the object when it is added to the cache and customized for the type of object being cached. An absolute expiration time is commonly used, and once the time elapses, the item is removed from the cache. The expiration time is chosen based on the client's requirements, such as how frequently the data change and how tolerant the system is to stale data. [20]

A sliding expiration policy is a way to determine how long a cached object should be retained, based on how frequently it is used by the application. This policy works by extending the expiration time of frequently accessed items by a specified interval each time they are accessed. This means that an item will not be removed from the cache as long as it is accessed within the specified time interval. The sliding expiration policy is a common way to invalidate cached objects, and it is often used to retain items that are frequently accessed by the application. [20]

When implementing a cache, it is important to choose an appropriate TTL value for each cached object. Once a TTL value is chosen, it is necessary to monitor the cache's effectiveness and re-evaluate the value if necessary. It should be noted that caching frameworks typically use a scavenging algorithm to remove expired items from the cache. This algorithm is typically invoked when referencing the cache, allowing for expired items to be flushed out efficiently without the need for constant tracking of expiration events. [20]

Other considerations

Here are some other general best practices for implementing caching in an application.

- Firstly, it is important to ensure that the data is cachable and that caching it will result in a high enough hit rate to justify the additional resources used.
- Monitoring the metrics of the caching infrastructure, such as hit rates and resource consumption, is also important to ensure that the cache is appropriately tuned, which can inform decisions regarding cache size, expiration, and eviction policies.
- The system should also be resilient to cache failure and scenarios such as cache unavailability, cache put/get failures, and downstream errors.
- The use of encryption techniques is recommended if sensitive data is retained in the cache, to mitigate security risks.
- Finally, the application should be designed to be resilient to changes in the storage format used for cached data, allowing new versions of the app to read data written by previous versions. [20]

3.1.6 Real-world examples of successful Node.js and PostgreSQL web applications with caching and load balancing?

Node.js and PostgreSQL are popular technologies used in building web applications, and many successful companies have implemented caching and load balancing to improve the performance and scalability of their systems.

Node.js

Netflix

Netflix embraced the adoption of Node.js as a means to facilitate large-scale web streaming to their extensive subscriber base, concurrently aiming to ensure observability, debuggability, and availability

of their underlying infrastructure. To effectively attain these objectives, the organization devised the NodeQuark infrastructure, leveraging an application gateway to authenticate and direct incoming requests towards the NodeQuark service. This service seamlessly communicates with APIs and processes responses, catering to the specific requirements of the clients. Consequently, this infrastructure empowers teams to develop tailored API experiences for diverse devices, thereby facilitating the seamless operation of the Netflix application across multiple platforms. [22]

Uber

Uber, a ride-hailing service, uses Node.js for its web application and database. To handle its massive user base and high traffic, Uber employs caching to speed up its app and reduce load on its servers. Load balancing is also used to distribute incoming requests across multiple servers, ensuring that the service remains fast and responsive even during periods of high demand. As a result, Uber can benefit from increased platform reliability, faster data processing, improved connectivity levels, and reduced overhead costs. [23]

LinkedIn

LinkedIn initially adopted Node.js for their mobile application and later transitioned their entire codebase to the platform. The company reported a 20 times faster app speed compared to their previous iteration built on Ruby on Rails. [24]

PostgreSQL

Apple

Apple, a leading technology company, has been utilizing PostgreSQL in its databases for a considerable amount of time. Apple replaced MySQL with Postgres as an embedded database in the OS X Lion release in 2010, and since then, Apple has focused on PostgreSQL due to product quality and concerns over changes in Oracle's MySQL licensing. Apple's systems currently support PostgreSQL, and it has become the default database on macOS Server since OS X Server version 10.7. Additionally, PostgreSQL is available in the App Store. [25]

Instagram

Instagram utilizes multiple relational database management systems (RDBMS) to support its operations. However, PostgreSQL and Cassandra were selected for the critical tasks. The objective was to minimize latency and guarantee seamless user experience on the platform. [25]

Twitch

The Twitch platform is built upon approximately 125 databases, with the majority of them being managed using PostgreSQL. These PostgreSQL databases include user databases, broadcast databases, and backup databases. [25]

These real-world examples demonstrate the effectiveness of caching and load balancing in improving the performance and scalability of Node.js and PostgreSQL web applications.

3.1.7 Challenges and trade-offs in implementing caching and load balancing in Node.js and PostgreSQL web applications?

Challenges of caching:

Cache invalidation

Cache invalidation is a process that removes outdated content from the cache, but it can be a challenge to decide when to do so. Purging the cache every time a change is made could harm website performance during high-traffic periods. Instead, invalidating the cache allows files to refresh in the background, ensuring visitors always see fast pages. Although this process may result in slightly outdated representations being served while new ones are prepared, it is a necessary step for effective web caching. [13]

Cache Consistency:

When using multiple cache instances, ensuring cache consistency across all instances can be a challenge. Inconsistent caching can lead to data inconsistencies, which can have a negative impact on user experience. [13]

Increased Memory Usage:

Caching requires additional memory to store cached data, which can be a challenge for systems with limited resources. This can lead to performance issues if the cache becomes too large, causing the system to slow down or crash. [13]

Trade-offs of caching:

Consistency vs. Performance:

Caching can lead to inconsistent data because the cached version may not always be up to date. This is a trade-off between consistency and performance. [13]

Cost vs. Performance:

Caching requires additional resources, such as memory and processing power. The trade-off between the cost of these resources and the performance benefits of caching must be considered. [13]

Increased Complexity:

Caching adds complexity to the application architecture, particularly in distributed environments. [13]

Load Balancing Challenges:

Server Failure:

Load balancing can be challenging if one or more servers fail, as the load must be redistributed to other servers. [14]

Overloaded Servers:

Load balancing must ensure that no server is overloaded with too much traffic, as this can affect performance. [14]

Session Persistence:

Session persistence, commonly referred to as sticky sessions, is a fundamental mechanism employed to ensure the consistent routing of client requests to a specific backend web or application server throughout the duration of a session. In this context, a session denotes the temporal period required to accomplish a given task or transaction. [26]

The utilization of load balancing techniques poses challenges to session persistence, as it involves the distribution of client requests across multiple backend servers. Consequently, when users navigate a website, they may be directed to different backend servers, thus jeopardizing the consistency of their interactions. This can result in complications if the web browser or backend server attempts to store information to expedite subsequent user actions. Therefore, ensuring that all requests from a user are consistently routed to the same server becomes paramount. The interference of load balancing with session persistence can significantly impact user experience and compromise the seamless functionality of applications. [27]

Trade-offs of load balancing:

Cost vs. Performance:

Load balancing requires additional resources, such as servers and networking equipment. The trade-off between the cost of these resources and the performance benefits of load balancing must be considered. [14]

Increased Complexity:

Load balancing adds complexity to the application architecture, particularly in distributed environments. [14]

Security vs. Performance:

Load balancing can affect the security of the application, particularly if sensitive data is being transmitted. The trade-off between security and performance must be considered. [14]

Session Hijacking and Cross-Site Scripting Attacks:

Load balancing can potentially lead to security issues such as session hijacking or cross-site scripting attacks. To mitigate these risks, various security measures such as SSL/TLS encryption, authentication, and authorization must be implemented. However, implementing these measures can increase the processing time and thus affect the performance of the application. [14]

In summary, implementing caching and load balancing in a Node.js and PostgreSQL web application can provide numerous benefits, but also presents challenges and trade-offs that must be considered. Careful planning and implementation can help ensure a well-performing and secure application.

3.2 Proof of Concept

3.2.1 High level overview

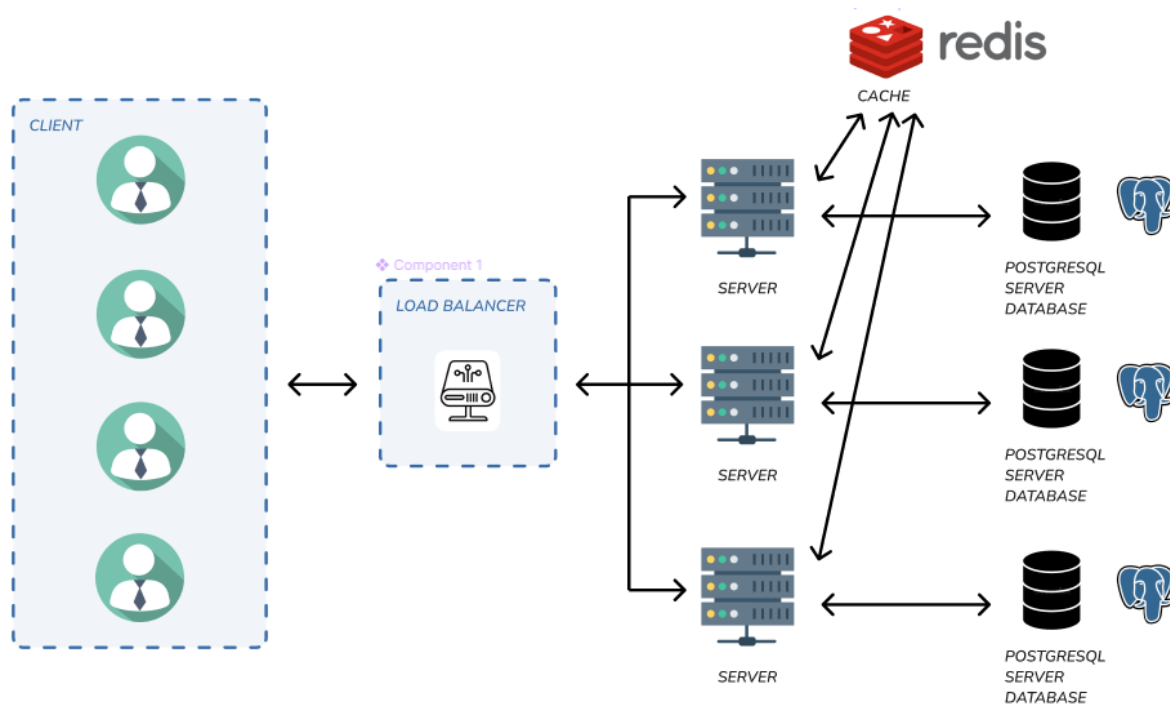


Figure 48: proof of concept overview

3.2.2 Set up Node.js & PostgreSQL project

Creating a PostgreSQL database

The initial step entails the installation of the PostgreSQL application and subsequent login to the user account. Following successful login, the creation of a role, designated as 'me', and assignment of the password 'password' is imperative. It is noteworthy that a role can fulfill the role of both a user and a group, and in this particular context, it will serve as a user. [28]

```
postgres=# CREATE ROLE me WITH LOGIN PASSWORD 'password';
CREATE ROLE
```

Figure 49: PostgreSQL CMD

Having established the role, the subsequent step entails the creation of a database through the utilization of the following SQL command. [28]

```
postgres=# CREATE DATABASE api;
CREATE DATABASE
postgres=# \list
```

Name	Owner	Encoding	Collate	List of databases Ctype	ICU Locale	Locale Provider	Access privileges
api	postgres	UTF8	English_Belgium.1252	English_Belgium.1252		libc	
postgres	postgres	UTF8	English_Belgium.1252	English_Belgium.1252		libc	
template0	postgres	UTF8	English_Belgium.1252	English_Belgium.1252		libc	=c/postgres +
template1	postgres	UTF8	English_Belgium.1252	English_Belgium.1252		libc	=c/postgres +

```
(4 rows)

postgres=# \c api
You are now connected to database "api" as user "postgres".
```

Figure 50: PostgreSQL CMD

After the successful creation of the database, the subsequent step involves the creation of a table named 'users' that comprises three distinct fields. This task can be accomplished by executing the necessary commands within the psql command prompt environment. [28]

```
postgres=# CREATE TABLE users (  
postgres(#   ID SERIAL PRIMARY KEY,  
postgres(#   name VARCHAR(30),  
postgres(#   email VARCHAR(30)  
postgres(# );  
CREATE TABLE
```

Figure 51: PostgreSQL CMD

In order to populate the 'users' table with data, it is necessary to execute the following SQL commands. [28]

```
postgres(# );  
CREATE TABLE  
postgres=# INSERT INTO users (name, email)  
postgres-#   VALUES ('Jerry', 'jerry@example.com'), ('George', 'george@example.com');  
INSERT 0 2  
postgres=# SELECT * FROM users;  
 id | name | email  
----+-----+-----  
  1 | Jerry | jerry@example.com  
  2 | George | george@example.com  
(2 rows)
```

Figure 52: PostgreSQL CMD

Having established the database and table, along with the populated data, the subsequent step involves constructing a Node.js RESTful API to establish connectivity with the PostgreSQL database. [28]

Setup Node.js project

To initialize a Node.js project, the first step is to create a directory and then execute the following command. [28]

```
npm init -y
```

Figure 53: CMD to setup project

Following the previous steps, the system will generate a file named 'package.json' to facilitate further configurations. Subsequently, the installation of the 'express' module for server implementation and the 'node-postgres' module for establishing a connection with PostgreSQL can be accomplished by executing the provided command. [28]

```
npm install express node-postgres
```

Figure 54: CMD to install dependency

Upon successfully installing the necessary dependencies and generating the 'package.json' file, the subsequent task involves creating an 'index.js' file to serve as the entry point for the server. At the beginning of the file, the 'express' module and the built-in 'body-parser' middleware are imported. Subsequently, the 'app' and 'port' variables are defined to establish the server configuration. [28]

```

1  const express = require('express')
2  const bodyParser = require('body-parser')
3  const app = express()
4  const port = 3000
5
6  app.use(bodyParser.json())
7  app.use(
8    bodyParser.urlencoded({
9      extended: true,
10    })
11  )

```

Figure 55: Setup project Node.js

A new route is introduced, specifically designed to deliver a response in the JSON format. [28]

```

13  app.get('/', (request, response) => {
14    response.json({ info: 'Node.js, Express, and Postgres API' })
15  })

```

Figure 56: New express path code

The application is configured to listen on the specified port. [28]

```

17  app.listen(port, () => {
18    console.log(`App running on port ${port}.`)
19  })

```

Figure 57: Finish setup project express

The server can be executed by issuing the following command. [28]

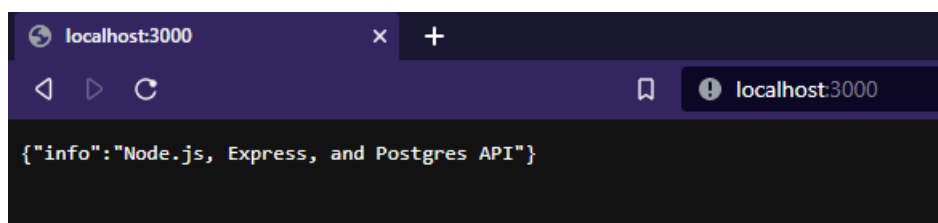
```

PS D:\Users\Explo\Documents\PXL\Internship Barcelona\PXL stage\Proof of concept app\node-api-postgres> node .\index.js
App running on port 3000.

```

Figure 58: CMD to run project

Accessing the URL <http://localhost:3000> will enable the retrieval of the previously defined JSON data. [28]



```

{"info": "Node.js, Express, and Postgres API"}

```

Figure 59: Retrieve data example

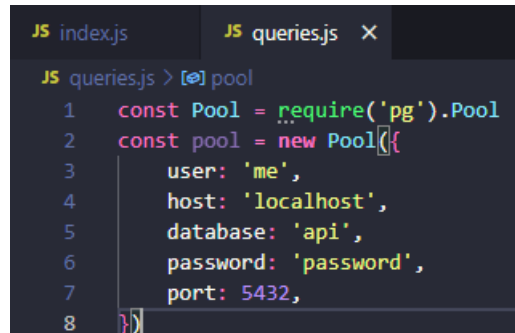
Following the successful execution of the server, the next step involves establishing a connection between Node.js and PostgreSQL. This connection is essential to facilitate the execution of dynamic queries. [28]

Connecting to a Postgres database from Node.js

To optimize connection management and eliminate the need to repeatedly open and close a client for each query, the 'node-postgres' module will be employed to establish a connection pool. For

production environments, 'pgBouncer', a lightweight connection pooler specifically designed for PostgreSQL, is a widely recommended option. [28]

To proceed, it is advised to create a file named 'queries.js' where the configuration for establishing a connection with PostgreSQL can be set up. [28]



```
JS index.js JS queries.js X
JS queries.js > pool
1 const Pool = require('pg').Pool
2 const pool = new Pool({
3   user: 'me',
4   host: 'localhost',
5   database: 'api',
6   password: 'password',
7   port: 5432,
8 })
```

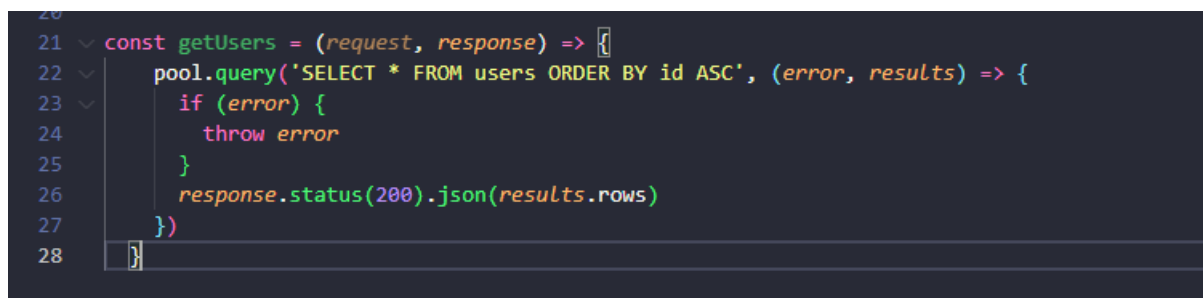
Figure 60: Configuration database

In a production environment, it is recommended to store configuration details in a separate file with restricted permissions to prevent unauthorized access through version control systems. However, for the sake of simplicity in this POC, the configuration details will be included in the same file as the queries. [28]

The purpose of this POC is to enable CRUD (Create, Read, Update, Delete) operations on the API, which will execute the corresponding database commands. This will involve setting up a route for each endpoint and implementing a corresponding function for each query. [28]

Creating routes for CRUD operations

The initial endpoint for this implementation will be a GET request, specifically designed to retrieve data. In this case, a 'SELECT' operation will be performed to fetch all users from the database, with the results ordered by the 'ID' column. [28]



```
20
21 const getUsers = (request, response) => {
22   pool.query('SELECT * FROM users ORDER BY id ASC', (error, results) => {
23     if (error) {
24       throw error
25     }
26     response.status(200).json(results.rows)
27   })
28 }
```

Figure 61: Retrieve all users code

To handle the user retrieval path, the custom ID parameter from the URL will be extracted, and a WHERE clause will be utilized to filter the query results based on this parameter. Notably, PostgreSQL adopts the numbered placeholder '\$1' instead of the more commonly recognized '?' placeholder found in other SQL variants for this specific case. [28]

```

30 const getUserById = (request, response) => {
31   const id = parseInt(request.params.id)
32
33   pool.query('SELECT * FROM users WHERE id = $1', [id], (error, results) => {
34     if (error) {
35       throw error
36     }
37     response.status(200).json(results.rows)
38   })
39 }

```

Figure 62: Retrieve user by id code

The API will support both GET and POST requests to the '/users' endpoint. For the POST request, it will facilitate the addition of a new user. Within this function, the name and email properties will be extracted from the request body, and the corresponding values will be inserted using the INSERT operation. [28]

```

41 const createUser = (request, response) => {
42   const { name, email } = request.body
43
44   pool.query('INSERT INTO users (name, email) VALUES ($1, $2) RETURNING *', [name, email], (error, results) => {
45     if (error) {
46       throw error
47     }
48     response.status(201).send(`User added with ID: ${results.rows[0].id}`)
49   })
50 }

```

Figure 63: Insert into users code

The '/users/:id' endpoint is designed to handle two distinct HTTP requests: GET, which has been previously implemented as 'getUserById', and PUT, utilized for modifying an existing user. To accomplish the user update functionality, the UPDATE clause will be employed, leveraging the knowledge acquired from both the GET and POST operations. [28]

It is important to note that the PUT method is idempotent, meaning that making the exact same call multiple times will yield the same result. This is in contrast to the POST method, where repeating the exact same call will create new users with the same data each time. [28]

```

52 const updateUser = (request, response) => {
53   const id = parseInt(request.params.id)
54   const { name, email } = request.body
55
56   pool.query(
57     'UPDATE users SET name = $1, email = $2 WHERE id = $3',
58     [name, email, id],
59     (error, results) => {
60       if (error) {
61         throw error
62       }
63       response.status(200).send(`User modified with ID: ${id}`)
64     }
65   )
66 }

```

Figure 64: Update user code

In conclusion, the 'DELETE' clause will be utilized on the '/users/:id' endpoint to enable the removal of a particular user based on their unique identifier. This operation closely resembles the previously implemented 'getUserById()' function. [28]

```
68 const deleteUser = (request, response) => {
69   const id = parseInt(request.params.id)
70
71   pool.query('DELETE FROM users WHERE id = $1', [id], (error, results) => {
72     if (error) {
73       throw error
74     }
75     response.status(200).send(`User deleted with ID: ${id}`)
76   })
77 }
```

Figure 65: Delete user code

Exporting CRUD functions in a REST API

To ensure the accessibility of the functions from the 'index.js' file, the 'module.exports' feature can be employed to export them. Through the creation of an object containing the functions, they can be exposed for utilization in other modules. [28]

```
68 module.exports = {
69   getUsers,
70   getUserById,
71   createUser,
72   updateUser,
73   deleteUser,
74 }
```

Figure 66: Export functions code

Setting up CRUD functions in a REST API

To integrate the queries into the 'index.js' file and configure endpoint routes for the created query functions, it is necessary to import the functions from the 'queries.js' module and assign them to a variable. [28]

```
5 const db = require('./queries')
```

Figure 67: Import code

Each endpoint will be defined by specifying the appropriate HTTP request method, URL path, and the corresponding function to be executed. This configuration ensures that the server can effectively handle incoming requests based on their characteristics. [28]

```
22 app.get('/users', db.getUsers)
23 app.get('/users/:id', db.getUserById)
24 app.post('/users', db.createUser)
25 app.put('/users/:id', db.updateUser)
26 app.delete('/users/:id', db.deleteUser)
```

Figure 68: Setup endpoints code

Upon the completion of the aforementioned files, the server setup, database connection, and API creation have been successfully implemented. To initiate the server, the identical command can be executed in the command prompt or terminal once more. [28]

```
PS D:\Users\Explo\Documents\PXL\Internship Barcelona\PXL stage\Proof of concept app\node-api-postgres> node .\index.js
App running on port 3000.
```

Figure 69: CMD to run the project

Upon accessing either 'http://localhost:3000/users' or 'http://localhost:3000/users/1' via a web browser or an HTTP client, the JSON responses generated by the respective GET requests can be observed. [28]

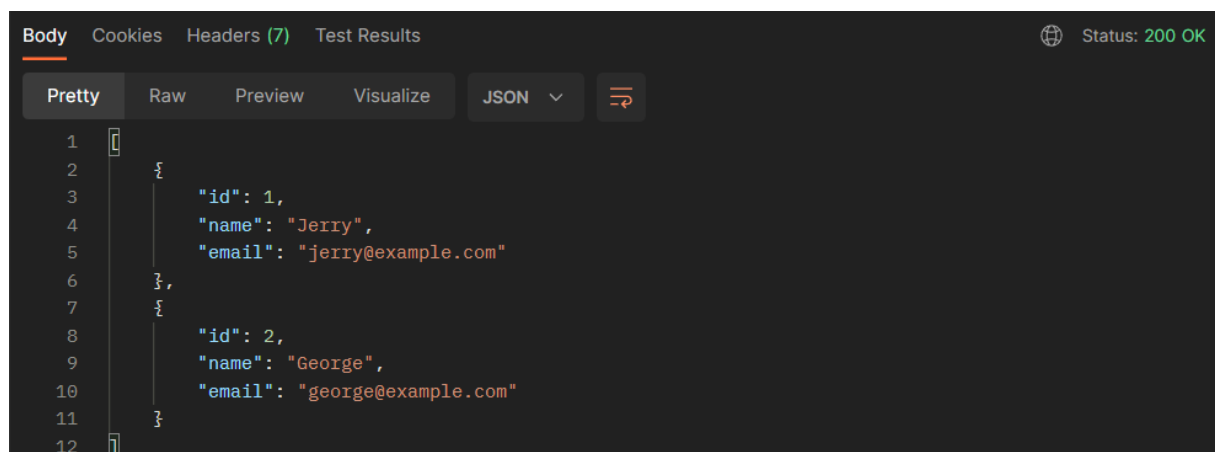


Figure 70: Postman GET request example

To validate the functionality of our POST, PUT, and DELETE requests, it is recommended to utilize tools such as Postman or Thunder Client, which facilitate the sending of HTTP requests. These tools simplify the process of querying endpoints with various HTTP methods. [28]

Alternatively, the curl command-line tool can be utilized to send HTTP requests. This command-line utility, commonly available in most terminals, enables the specification of the URL, the desired HTTP method, and any additional parameters for request customization. [28]

Utilizing these tools facilitates the evaluation of the API endpoints' functionality and facilitates the observation of the corresponding generated responses. [28]

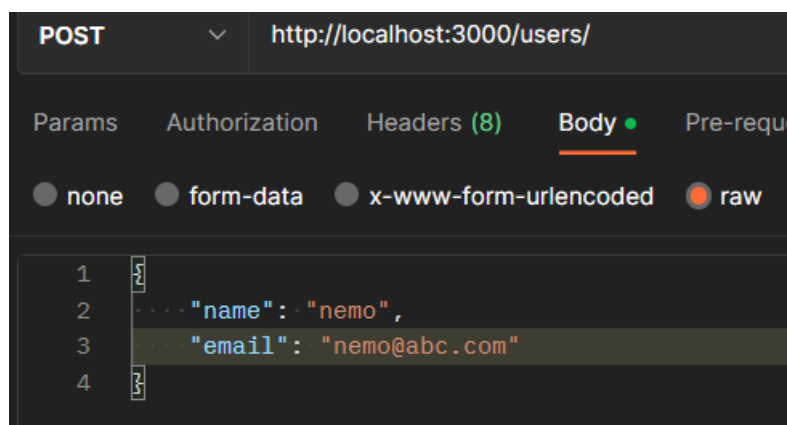


Figure 71: Postman add user example

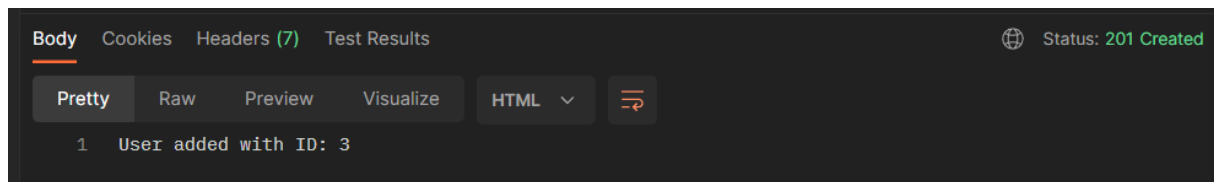


Figure 72: Postman add user output

3.2.3 Set up caching

To integrate the Redis framework into the Node.js project, execute the specified command within the Windows Subsystem for Linux (WSL) environment. [29]

```
npm install express redis axios
```

Figure 73: Install dependency

To optimize the utilization of Redis, it is advised to compile the Redis source code directly. This can be achieved by executing the provided command in the terminal. [29]

```
wget http://download.redis.io/redis-stable.tar.gz
tar xvzf redis-stable.tar.gz
cd redis-stable
make
```

Figure 74: CMD to install Redis

The successful installation of Redis can be confirmed by executing the provided command. [29]

```
make test
```

Figure 75: CMD to confirm installation of Redis

To add Redis to your system's path, you can execute the given command. [29]

```
sudo make install
```

Figure 76: CMD to copy Redis to path

To confirm the successful setup of Redis, you can initiate the Redis server by executing the provided command in your terminal. [29]

```
redis-server
```

Figure 77: CMD to execute the Redis server

Subsequently, you can open a new terminal tab or window and execute the provided command. [29]

```
redis-cli ping
```

Figure 78: CMD to indicate Redis is running successfully

Upon executing the command, a response of "PONG" confirms the successful setup of Redis. [29]

```
drake@LAPTOP-EA9RQKP0:~$ redis-cli ping
PONG
```

Figure 79: Ping output example

With Redis now configured, the application can leverage its capabilities to enhance performance by reducing the latency between requests and responses. To implement these improvements, the following changes should be incorporated into the 'index.js' file. [29]

```
7   port = 3452;
8   })
9   const redis = require('redis');
10
11   let redisClient;
12
13   (async () => {
14     redisClient = redis.createClient();
15
16     redisClient.on("error", (error) => console.error(`Error : ${error}`));
17
18     await redisClient.connect();
19   })();
20
21   const getUsers = (request, response) => {
```

Figure 80: Import Redis code

The implementation of caching for the 'getUserById' method will now be undertaken. [29]

```
30   const getUserById = async (request, response) => {
31     const id = request.params.id;
32     let isCached = false;
33
34     // Check the Redis cache first
35     const cacheResults = await redisClient.get(id);
36     if (cacheResults) {
37       // If user data is in the cache
38       isCached = true;
39       return response.status(200).json(JSON.parse(cacheResults));
40     } else {
41       // If user data is not in the cache, query the database
42       pool.query('SELECT * FROM users WHERE id = $1', [id], async (error, results) => {
43         if (error) {
44           throw error;
45         }
46
47         // Store the user data in the Redis cache for future requests
48         await redisClient.set(id, JSON.stringify(results.rows[0]));
49
50         return response.status(200).json(results.rows[0]);
51       });
52     }
53   };
```

Figure 81: Implement caching on endpoint code

```
drake@LAPTOP-EA9RQKP0:~/NodeExpressAPI$ time curl http://localhost:3000/users/1
{"id":1,"name":"Jerry","email":"jerry@example.com"}
real    0m0.029s
user    0m0.002s
sys     0m0.006s
```

Figure 82: GET request example

After the implementation of caching, an evaluation of its impact on performance can be conducted. In the case of a GET request for a user, it is noted that the retrieval process requires approximately 29 milliseconds. This duration is attributed to the absence of the requested user data in the cache, necessitating the retrieval from the database. [29]

```
drake@LAPTOP-EA9RQKP0:~/NodeExpressAPI$ time curl http://localhost:3000/users/1
{"id":1,"name":"Jerry","email":"jerry@example.com"}
real    0m0.012s
user    0m0.009s
sys     0m0.001s
```

Figure 83: GET request example repeated

Upon subsequent execution of the identical GET request, a discernible decrease in response time to approximately 12 milliseconds is evident. This notable improvement in response time can be attributed to the efficient retrieval of data from the cache, thus circumventing the need for a database query. [29]

3.2.4 Set up Siege

Installation

The decision has been made to utilize Siege, a load testing tool, to perform a comprehensive assessment of the application's performance under simulated high traffic conditions. Siege, developed by Jeff Fulmer, is specifically designed for HTTP load testing and benchmarking purposes. The tool can be conveniently installed on MacOS through the Homebrew package manager, while on various Linux distributions, it is readily accessible from the default repository. This installation process is typically swift and straightforward, facilitating seamless adoption of the tool for testing and analysis purposes. [30]

As a preliminary step, it is necessary to install Siege by executing the following command in the WSL. [30]

```
sudo apt install siege
```

Figure 84: install siege CMD

The following command will be employed to execute the load tests: [30]

```
siege --delay=0.5 --file=urls.txt --internet --verbose --reps=1000 --concurrent=500
```

Figure 85: Siege test CMD

The "delay" setting introduces a random delay between each client's request, ranging from zero to the specified number of seconds (in this case, 0.5 seconds). This deliberate delay adds a sense of realism to the test by preventing all clients from simultaneously bombarding the server. Notably, Siege conveniently excludes the delay from the output, eliminating concerns of skewing the test results. [30]

The "file" setting allows the specification of a file containing a collection of URLs to be utilized in the test. This facilitates the systematic testing of multiple URLs in a controlled manner. [30]

The "internet" setting randomizes the selection of URLs from the specified file, providing variation and avoiding biased patterns in the test. [30]

The "no-parser" option, when enabled, prevents Siege from parsing the returned HTML and requesting associated assets such as CSS, JavaScript, and images. This is particularly relevant in scenarios where production assets are served from a CDN, rendering the test more representative of real-world conditions. [30]

The "reps" setting determines the number of times the test will be repeated, allowing for multiple iterations to gather sufficient data and assess the system's performance consistently. [30]

The "concurrent" setting specifies the number of clients (simulated users) participating simultaneously in the test, enabling the simulation of varying levels of concurrent user activity. [30]

Finally, the "verbose" setting generates detailed information about each request as the test runs, providing valuable insights into the test execution and aiding in result analysis. [30]

The "file" parameter refers to a simple file containing a list of URLs, with each URL occupying a separate line. This parameter serves the purpose of defining the endpoints that will be used in the testing process. [30]

By configuring these settings appropriately, the load testing process can be conducted effectively, providing valuable performance metrics and insights into the tested system's behavior.

3.2.5 Load testing caching

Load testing

In order to evaluate the impact of caching on the performance of the application, the caching mechanism was intentionally disabled or removed. Subsequently, a series of tests were conducted to assess the application's performance under these conditions. By comparing the results of these tests to the previous ones conducted with caching enabled, significant insights were obtained regarding the influence of caching on the responsiveness and overall efficiency of the application.

In order to evaluate the performance of our application subsequent to the integration of caching, a thorough and systematic series of tests was executed. These tests were specifically designed to encompass a predefined set of endpoints, meticulously generated utilizing a shell script file. The sequential generation of endpoints, ranging from 1 to 160, provided a comprehensive and diverse range of test scenarios to assess the overall functionality and responsiveness of our application. By employing the shell script file, the process of generating the desired endpoints was expedited, optimizing the efficiency and effectiveness of our testing methodology.


```
drake@LAPTOP-EA9RQKP0:~/downloads$ cat urls.txt
http://localhost:3000/users/1
http://localhost:3000/users/2
http://localhost:3000/users/3
http://localhost:3000/users/4
http://localhost:3000/users/5
http://localhost:3000/users/6
http://localhost:3000/users/7
http://localhost:3000/users/8
http://localhost:3000/users/9
http://localhost:3000/users/10
http://localhost:3000/users/11
http://localhost:3000/users/12
```

Figure 86: content file parameter

A comprehensive and meticulous series of tests was conducted to meticulously analyze the performance and responsiveness of the application. These rigorous tests yielded invaluable insights into the application's behavior and its capability to handle diverse scenarios under varying conditions. The findings obtained from these tests facilitated a comprehensive understanding of the application's performance characteristics, enabling us to make informed decisions and optimizations to enhance its overall efficiency and user experience.

First test

```
drake@LAPTOP-EA9RQKP0:~/downloads$ siege --delay=0.5 --file=urls.txt --internet --verbose --reps=1000 --concurrent=500
{
  "transactions":          500000,
  "availability":          100.00,
  "elapsed_time":          266.27,
  "data_transferred":      26.71,
  "response_time":         0.00,
  "transaction_rate":      1877.79,
  "throughput":            0.10,
  "concurrency":           3.40,
  "successful_transactions": 500000,
  "failed_transactions":    0,
  "longest_transaction":    0.57,
  "shortest_transaction":   0.00
}
```

Figure 87: first test with caching

```
drake@LAPTOP-EA9RQKP0:~/downloads$ siege --delay=0.5 --file=urls.txt --internet --verbose --reps=1000 --concurrent=500
{
  "transactions":          500000,
  "availability":          100.00,
  "elapsed_time":          265.13,
  "data_transferred":      26.71,
  "response_time":         0.00,
  "transaction_rate":      1885.87,
  "throughput":            0.10,
  "concurrency":           3.99,
  "successful_transactions": 500000,
  "failed_transactions":    0,
  "longest_transaction":    0.37,
  "shortest_transaction":   0.00
}
```

Figure 88: first test without caching and load balancing

Second test

```
drake@LAPTOP-EA9RQKP0:~/downloads$ siege --delay=0.5 --file=urls.txt --internet --verbose --reps=1500 --concurrent=750

{
  "transactions":          1125000,
  "availability":          100.00,
  "elapsed_time":          397.62,
  "data_transferred":      60.10,
  "response_time":          0.00,
  "transaction_rate":      2829.33,
  "throughput":             0.15,
  "concurrency":           10.47,
  "successful_transactions": 1125000,
  "failed_transactions":    0,
  "longest_transaction":    1.41,
  "shortest_transaction":   0.00
}
```

Figure 89: second test with caching

```
drake@LAPTOP-EA9RQKP0:~/downloads$ siege --delay=0.5 --file=urls.txt --internet --verbose --reps=1500 --concurrent=750

{
  "transactions":          1125000,
  "availability":          100.00,
  "elapsed_time":          408.49,
  "data_transferred":      60.09,
  "response_time":          0.01,
  "transaction_rate":      2754.05,
  "throughput":             0.15,
  "concurrency":           36.78,
  "successful_transactions": 1125000,
  "failed_transactions":    0,
  "longest_transaction":    0.81,
  "shortest_transaction":   0.00
}
```

Figure 90: second test without caching and load balancing

Third test

```
drake@LAPTOP-EA9RQKP0:~/downloads$ siege --delay=0.5 --file=urls.txt --internet --verbose --reps=2000 --concurrent=1000

{
  "transactions":          2000000,
  "availability":          100.00,
  "elapsed_time":          585.36,
  "data_transferred":      106.84,
  "response_time":          0.03,
  "transaction_rate":      3416.70,
  "throughput":             0.18,
  "concurrency":           110.38,
  "successful_transactions": 2000000,
  "failed_transactions":    0,
  "longest_transaction":    0.85,
  "shortest_transaction":   0.00
}
```

Figure 91: third test with caching

```
drake@LAPTOP-EA9RQKP0:~/downloads$ siege --delay=0.5 --file=urls.txt --internet --verbose --reps=2000 --concurrent=1000

{
  "transactions":          2000000,
  "availability":          100.00,
  "elapsed_time":          719.47,
  "data_transferred":      106.84,
  "response_time":          0.10,
  "transaction_rate":      2779.82,
  "throughput":             0.15,
  "concurrency":           282.19,
  "successful_transactions": 2000000,
  "failed_transactions":    0,
  "longest_transaction":    1.01,
  "shortest_transaction":   0.05
}
```

Figure 92: third test without caching and load balancing

Conclusion

The provided load testing results demonstrate the notable differences between an application without caching and the same application with caching enabled. These differences highlight the significant benefits of incorporating caching into web applications.

Response Time: The introduction of caching leads to a substantial reduction in response time. Without caching, the reported response time is 0.10 seconds. However, with caching enabled, the response time decreases to 0.03 seconds. This considerable improvement indicates that cached data can be accessed more quickly, resulting in faster retrieval and delivery of content to users.

Transaction Rate: The transaction rate, which measures the number of transactions processed per unit of time, demonstrates an increase with caching. Without caching, the transaction rate is reported as 2779.82 transactions per second. However, with caching enabled, it rises to 3416.70 transactions per second. This higher transaction rate signifies that caching allows the application to handle a larger volume of requests and process them more efficiently, resulting in improved scalability.

Concurrency: The concurrency value represents the number of concurrent users or connections that the application can handle simultaneously. In the absence of caching, the concurrency value is 282.19. However, with caching enabled, it reduces to 110.38. This decrease in concurrency indicates that caching reduces the load on the application server, enabling it to handle a higher number of concurrent connections without experiencing performance degradation.

Elapsed Time: The elapsed time, which measures the total time taken to complete the load testing, decreases with caching. Without caching, the elapsed time is 719.47 seconds. However, with caching enabled, it decreases to 585.36 seconds. This reduction in elapsed time demonstrates that caching enhances the overall efficiency of the application by reducing the time required to process and deliver content.

Based on these results, it can be concluded that the implementation of caching in web applications offers several benefits. These include:

Improved Performance: Caching significantly reduces response times, resulting in faster retrieval and delivery of content. This ultimately enhances the user experience and increases user satisfaction.

Enhanced Scalability: Caching enables the application to handle a larger number of transactions per unit of time, thereby improving scalability. It allows the system to accommodate more concurrent users or connections without compromising performance.

Reduced Server Load: Caching reduces the concurrency value, indicating a lighter load on the application server. This leads to improved server utilization and resource efficiency.

Increased Throughput: With caching, there is a slight increase in throughput, representing the amount of data transferred per unit of time. This suggests that caching optimizes resource utilization and enhances the overall throughput of the application.

In conclusion, caching plays a critical role in optimizing system performance by reducing response times, minimizing server load, and improving scalability. By efficiently storing and retrieving frequently accessed data, caching significantly enhances the user experience, enables the application

to handle more concurrent requests, and reduces the overall time required to process and deliver content.

3.2.6 Set up load balancing

Load balancing can be achieved through various means, with Nginx being a popular option. To set up Nginx for load balancing, begin by installing it on your system and configuring it as a reverse proxy. This involves editing the Nginx configuration file. Below is an example configuration file: [31]

```
events {
    worker_connections 1024;
}

http {
    upstream backend {
        server localhost:3000;
        server localhost:3001;
        server localhost:3002;
    }

    server {
        listen 80;
        server_name localhost;

        location / {
            proxy_pass http://backend;
        }
    }
}
```

Figure 93: Nginx configuration file

This Nginx configuration file is specifically designed to implement load balancing for a Node.js application. The events block specifies the maximum number of connections allowed for each worker process. The 'http' block defines an upstream group of servers which will receive incoming requests and be load balanced across. The server block listens for requests on port 80 and passes them to the upstream group of servers defined in the 'http' block. Lastly, the location block is used to specify the URL path that requests should be passed to, which in this case, directs them to the backend group of servers defined earlier in the configuration file. [31]

To enable Nginx to load balance requests across multiple backend servers, multiple instances of the Node.js application must be started, each running on a different port. The cluster module in Node.js can be utilized to create multiple worker processes, each listening on different ports. The index.js file must be edited accordingly to enable this. [31]

```

1  const cluster = require('cluster');
2  const express = require('express');
3  const bodyParser = require('body-parser');
4  const db = require('./queries');
5
6  const app = express();
7  const port = 3000;
8
9  app.use(bodyParser.json());
10 app.use(bodyParser.urlencoded({
11     extended: true
12 }));
13
14 app.get('/', (request, response) => {
15     response.json({ info: 'Node.js, Express, and Postgres API' })
16 })
17
18 app.get('/users', db.getUsers)
19 app.get('/users/:id', db.getUserById)
20 app.post('/users', db.createUser)
21 app.put('/users/:id', db.updateUser)
22 app.delete('/users/:id', db.deleteUser)
23
24 if (cluster.isMaster) {
25     const numCPUs = require('os').cpus().length;
26     for (let i = 0; i < numCPUs; i++) {
27         cluster.fork();
28     }
29 } else {
30     app.listen(port, () => {
31         console.log(`App running on port ${port}.`)
32     })
33 }

```

Figure 94: Setup load balancing

In this code, the cluster module is used to fork multiple worker processes that listen on port 3000.

Once Nginx has been properly configured and multiple instances of the Node.js application have been initiated, Nginx can be launched to facilitate load balancing of incoming requests targeting the backend servers. The commencement of Nginx can be achieved by executing the subsequent command: [31]

```
sudo service nginx start
```

Figure 95: CMD to run Nginx

Upon successfully launching Nginx, requests can be dispatched to the Node.js application by accessing the Nginx server through port 80. Nginx will effectively distribute incoming requests across

the designated backend servers as specified in the configuration file, thereby facilitating load balancing functionality. [31]

```
drake@LAPTOP-EA9RQKP0:~/NodeExpressAPI$ time curl http://localhost:80/users/7
{"id":7,"name":"Estelle","email":"estelle@example.com"}
real    0m0.070s
user    0m0.010s
sys     0m0.000s
drake@LAPTOP-EA9RQKP0:~/NodeExpressAPI$ time curl http://localhost:80/users/7
{"id":7,"name":"Estelle","email":"estelle@example.com"}
real    0m0.023s
user    0m0.010s
sys     0m0.000s
drake@LAPTOP-EA9RQKP0:~/NodeExpressAPI$ time curl http://localhost:80/users/9
{"id":9,"name":"Susan","email":"susan@example.com"}
real    0m0.037s
user    0m0.010s
sys     0m0.000s
drake@LAPTOP-EA9RQKP0:~/NodeExpressAPI$ time curl http://localhost:80/users/9
{"id":9,"name":"Susan","email":"susan@example.com"}
real    0m0.026s
user    0m0.011s
sys     0m0.000s
```

Figure 96: Get requests example

3.2.7 Load testing balancing

The ports in the "urls.txt" file were modified to 80 through the utilization of a shell script file.

First test

```
drake@LAPTOP-EA9RQKP0:~/downloads$ siege --delay=0.5 --file=urls.txt --internet --verbose --reps=1000 --concurrent=500
{
  "transactions":          500000,
  "availability":          100.00,
  "elapsed_time":          266.56,
  "data_transferred":      66.84,
  "response_time":         0.00,
  "transaction_rate":      1875.75,
  "throughput":            0.25,
  "concurrency":           4.24,
  "successful_transactions": 500000,
  "failed_transactions":    0,
  "longest_transaction":    0.61,
  "shortest_transaction":   0.00
}
```

Figure 97: first with load balancing

```
drake@LAPTOP-EA9RQKP0:~/downloads$ siege --delay=0.5 --file=urls.txt --internet --verbose --reps=1000 --concurrent=500
{
  "transactions":          500000,
  "availability":          100.00,
  "elapsed_time":          265.13,
  "data_transferred":      26.71,
  "response_time":         0.00,
  "transaction_rate":      1885.87,
  "throughput":            0.10,
  "concurrency":           3.99,
  "successful_transactions": 500000,
  "failed_transactions":    0,
  "longest_transaction":    0.37,
  "shortest_transaction":   0.00
}
```

Figure 98: first test without caching and load balancing

Second test

```
drake@LAPTOP-EA9RQKP0:~/downloads$ siege --delay=0.5 --file=urls.txt --internet --verbose --reps=1500 --concurrent=750
{
  "transactions":          1125000,
  "availability":          100.00,
  "elapsed_time":          398.05,
  "data_transferred":      60.09,
  "response_time":         0.00,
  "transaction_rate":      2826.28,
  "throughput":            0.15,
  "concurrency":           8.42,
  "successful_transactions": 1125000,
  "failed_transactions":    0,
  "longest_transaction":   1.22,
  "shortest_transaction":  0.00
}
```

Figure 99: second test with load balancing

```
drake@LAPTOP-EA9RQKP0:~/downloads$ siege --delay=0.5 --file=urls.txt --internet --verbose --reps=1500 --concurrent=750
{
  "transactions":          1125000,
  "availability":          100.00,
  "elapsed_time":          408.49,
  "data_transferred":      60.09,
  "response_time":         0.01,
  "transaction_rate":      2754.05,
  "throughput":            0.15,
  "concurrency":           36.78,
  "successful_transactions": 1125000,
  "failed_transactions":    0,
  "longest_transaction":   0.81,
  "shortest_transaction":  0.00
}
```

Figure 100: second test without caching and load balancing

Third test

```
drake@LAPTOP-EA9RQKP0:~/downloads$ siege --delay=0.5 --file=urls.txt --internet --verbose --reps=2000 --concurrent=1000
{
  "transactions":          2000000,
  "availability":          100.00,
  "elapsed_time":          543.25,
  "data_transferred":      106.84,
  "response_time":         0.01,
  "transaction_rate":      3681.55,
  "throughput":            0.20,
  "concurrency":           41.02,
  "successful_transactions": 2000000,
  "failed_transactions":    0,
  "longest_transaction":   1.92,
  "shortest_transaction":  0.00
}
```

Figure 101: third test with load balancing

```
drake@LAPTOP-EA9RQKP0:~/downloads$ siege --delay=0.5 --file=urls.txt --internet --verbose --reps=2000 --concurrent=1000
{
  "transactions": 2000000,
  "availability": 100.00,
  "elapsed_time": 719.47,
  "data_transferred": 106.84,
  "response_time": 0.10,
  "transaction_rate": 2779.82,
  "throughput": 0.15,
  "concurrency": 282.19,
  "successful_transactions": 2000000,
  "failed_transactions": 0,
  "longest_transaction": 1.01,
  "shortest_transaction": 0.05
}
```

Figure 102: third test without caching and load balancing

Conclusion

Load Test with 1000 Concurrent Users and 2000 Repetitions:

Elapsed Time: The presence of load balancing reduced the completion time from 719.47 seconds to 543.25 seconds, resulting in a significant improvement of 176.22 seconds.

Response Time: Load balancing led to a substantial decrease in response time, reducing it from 0.10 seconds to 0.01 seconds, thereby enhancing the user experience and improving application responsiveness.

Transaction Rate: Load balancing showcased notable advantages, increasing the transaction rate from 2779.82 transactions per second to 3681.55 transactions per second, indicating improved throughput and increased capacity to handle a higher volume of transactions within the same time frame.

Load Test with 500 Concurrent Users and 1000 Repetitions:

Elapsed Time: The difference in completion times between load balancing and no load balancing was negligible, with completion times of 265.13 seconds and 267.07 seconds, respectively. Load balancing did not significantly improve elapsed time for smaller workloads with fewer users and repetitions.

Response Time: Both load-balanced and non-load-balanced configurations maintained consistently low response times of 0.00 seconds, indicating that load balancing did not have a substantial impact on response time in this scenario.

Transaction Rate: Load balancing resulted in a slightly lower transaction rate of 1872.17 transactions per second compared to 1885.87 transactions per second without load balancing. However, the transaction rate remained high for both cases, indicating efficient transaction handling in this workload.

Load Test with 750 Concurrent Users and 1500 Repetitions:

Elapsed Time: Load balancing demonstrated notable performance improvements, reducing the completion time from 408.49 seconds to 398.05 seconds, highlighting a noticeable improvement in completing the load testing.

Response Time: Load balancing maintained a consistently low response time of 0.00 seconds, even with higher concurrency, showcasing its ability to handle increased user demands without compromising performance.

Transaction Rate: Load balancing resulted in an improved transaction rate of 2826.28 transactions per second compared to 2754.05 transactions per second without load balancing, indicating enhanced throughput and increased processing capacity.

In conclusion, load balancing proves highly advantageous for larger workloads with higher numbers of concurrent users. It significantly reduces elapsed time, enhances response time, and improves transaction rates, thereby augmenting throughput and providing an optimized user experience. However, for smaller workloads comprising fewer users and repetitions, the benefits of load balancing may not be as pronounced. Nevertheless, load balancing contributes to stability, resource utilization, and future scalability of the application, establishing a solid foundation for efficiently handling larger workloads when necessary.

Conclusion

This research paper delved into the key aspects of designing scalable web applications with Node.js and PostgreSQL, focusing on caching and load balancing as crucial components. Node.js emerged as a powerful platform for web application development, offering an event-driven architecture, asynchronous programming model, and a wide range of use cases across industries. Its seamless integration with PostgreSQL, a stable and versatile open-source database system, further enhanced the scalability and reliability of web applications.

The exploration of caching emphasized its importance in optimizing system performance, reducing latency, and alleviating the load on the database. Various caching techniques were discussed, including in-memory caching, database caching, web client caching, web server caching, and CDN caching. Each caching strategy showcased its benefits and appropriate use cases, empowering developers to make informed decisions based on their specific requirements.

Load balancing was identified as a fundamental practice for achieving scalability and improving various aspects of website performance. Load balancers were recognized as valuable assets in distributing network traffic efficiently, enhancing security, optimizing performance, ensuring resilience, and enabling seamless scalability.

While implementing caching and load balancing, developers should be aware of the challenges and trade-offs involved. Cache invalidation, cache consistency, server failure, overloaded servers, session persistence, and resource allocation are among the factors that require careful consideration. However, with proper planning and implementation, these challenges can be overcome, allowing developers to strike a balance between performance, scalability, cost, consistency, and security.

Real-world examples from successful companies demonstrated the effectiveness of caching and load balancing techniques in improving application speed, reliability, data processing, and connectivity levels. By following best practices for caching and load balancing, developers can create highly performant and scalable web applications with Node.js and PostgreSQL, ultimately delivering a superior user experience.

Caching plays a critical role in optimizing system performance by reducing response times, minimizing server load, and improving scalability. By efficiently storing and retrieving frequently accessed data, caching significantly enhances the user experience, enables the application to handle more concurrent requests, and reduces the overall time required to process and deliver content.

Load balancing proves highly advantageous for larger workloads with higher numbers of concurrent users. It significantly reduces elapsed time, enhances response time, and improves transaction rates, thereby augmenting throughput and providing an optimized user experience. However, for smaller workloads comprising fewer users and repetitions, the benefits of load balancing may not be as pronounced. Nevertheless, load balancing contributes to stability, resource utilization, and future scalability of the application, establishing a solid foundation for efficiently handling larger workloads when necessary.

In conclusion, the combination of Node.js, PostgreSQL, caching, and load balancing offers a robust foundation for building scalable web applications. By leveraging the architectural strengths of Node.js, the versatility of PostgreSQL, the optimization capabilities of caching, and the traffic distribution of load balancing, developers can create high-performance web applications capable of handling concurrent requests, maintaining data integrity, and delivering an exceptional user experience.

III. Reflection

During the internship at Lemonade, a deeper understanding of React and other technologies was acquired, accompanied by the improvement of coding practices through feedback obtained during pull requests. Colleague input aided in the refinement of coding practices and the adoption of software development best practices. The value of code reviews was recognized as an opportunity for growth and refinement, and the assimilation of suggested improvements was actively pursued.

Throughout the project, various tickets presented implementation challenges, including uncertainties regarding the integration of specific features like the OAuth system. To overcome these challenges, extensive research and examination of successful implementations from other projects were conducted. However, seeking guidance from a teammate remained essential to effectively implement the system and address associated obstacles.

Another challenge arose when incorporating visible query parameters into the URL. Although multiple online solutions were available, integration into the existing application code remained uncertain. Consequently, assistance from a teammate was sought, leading to code review, issue identification, and guidance on the correct implementation of query parameters. Their assistance proved vital in resolving the challenge and ensuring successful functionality integration.

The collaborative environment within the team facilitated seeking help and learning from others' expertise, promoting personal and professional growth. Recognizing that seeking assistance when confronted with challenges is a sign of strength, hurdles were overcome, and problem-solving skills were enhanced. The supportive and collaborative atmosphere at Lemonade contributed to these achievements.

The implementation process adhered to a tailored Scrum methodology characterized by 6-week release cycles, which aimed to optimize efficiency and ensure timely software delivery. Daily scrum meetings played a crucial role in synchronizing efforts, evaluating progress, and identifying any obstacles. These meetings provided a platform for alignment, adaptability, and cohesive workflows. Supervised by Johan Maes, progress was actively monitored, and necessary adjustments were made to ensure project success and timely completion.

By leveraging available knowledge and guidance, all assigned tickets were successfully addressed while maintaining focus on project objectives. The migration of the Angular application to React resulted in a visually enhanced version, demonstrating dedication, perseverance, and the support received from colleagues.

This internship experience at Lemonade provided not only technical skills and knowledge but also a sense of accomplishment and pride in the work performed. The significance of collaboration, continuous improvement, and seeking assistance when necessary became apparent in achieving success. The ability to adapt, learn, and produce high-quality outcomes constitutes a valuable lesson to be carried forward into future endeavors.

In conclusion, the internship at Lemonade proved to be an enriching experience, encompassing the acquisition of React knowledge, collaboration with backend packages, and the valuable feedback obtained through pull requests. The importance of collaboration and seeking assistance when confronted with challenges was embraced. Gratitude is expressed for the opportunity to contribute to the creation of an improved version of the Angular application and for the supportive and collaborative environment fostering growth as a software developer. This experience has further

fueled a passion for software development and provided a solid foundation for future professional pursuits.

4 Bibliographical references

- [1] Node.js, "About," Node.js, [Online]. Available: <https://nodejs.org/en/about>. [Accessed 24 03 2023].
- [2] Geeksforgeeks, "Thread in OS," Geeksforgeeks, [Online]. Available: <https://www.geeksforgeeks.org/thread-in-operating-system/>. [Accessed 24 03 2023].
- [3] "The Node.js Event Loop, Timers, and process.nextTick()," Node.js, [Online]. Available: <https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick>. [Accessed 26 5 2023].
- [4] Node.js, "Blocking vs non blocking," Node.js, [Online]. Available: <https://nodejs.org/en/docs/guides/blocking-vs-non-blocking>. [Accessed 24 03 2024].
- [5] "What is node js?," Kinsta, [Online]. Available: <https://kinsta.com/knowledgebase/what-is-node-js/>. [Accessed 24 03 2023].
- [6] M. Gupta, "Node.js basic architecture," Medium, [Online]. Available: <https://medium.com/technofunnel/node-js-single-threaded-event-based-architecture-9f73daee37a1>. [Accessed 31 03 2023].
- [7] Kinsta, "What is postgresql?," Kinsta, [Online]. Available: <https://kinsta.com/knowledgebase/what-is-postgresql/>. [Accessed 31 03 2023].
- [8] Awardspace, "What is PostgreSQL and How Is It Used in Web Hosting?," Awardspace, [Online]. Available: <https://www.awardspace.com/kb/what-is-postgresql/>. [Accessed 23 4 2023].
- [9] "SQLite vs MySQL vs PostgreSQL: A Comparison Of Relational Database Management Systems," Digital Ocean, [Online]. Available: <https://www.digitalocean.com/community/tutorials/sqlite-vs-mysql-vs-postgresql-a-comparison-of-relational-database-management-systems>. [Accessed 26 5 2023].
- [10] I. Panchenko, "PostgreSQL benefits and challenges: A snapshot," Infoworld, [Online]. Available: <https://www.infoworld.com/article/3619531/postgresql-benefits-and-challenges-a-snapshot.html>. [Accessed 26 5 2023].
- [11] K. Hristozov, "MySQL vs PostgreSQL -- Choose the Right Database for Your Project," okta developer, [Online]. Available: <https://developer.okta.com/blog/2019/07/19/mysql-vs-postgres>. [Accessed 26 5 2023].
- [12] "ADVANTAGES OF POSTGRESQL," Cybertec, [Online]. Available: <https://www.cybertec-postgresql.com/en/postgresql-overview/advantages-of-postgresql/>. [Accessed 26 5 2023].
- [13] Auth0, "What is Caching and How It Works," Auth0, [Online]. Available: <https://auth0.com/blog/what-is-caching-and-how-it-works/>. [Accessed 31 3 2023].
- [14] R. Pescow, "What Is Overhead? What Small Businesses Need to Know," [Online]. Available: <https://www.nerdwallet.com/article/small-business/what-is-overhead-what-small-businesses->

need-to-know. [Accessed 8 6 2023].

- [15] Nginx, "What is load balancing?," Nginx, [Online]. Available: <https://www.nginx.com/resources/glossary/load-balancing/>. [Accessed 9 4 2023].
- [16] Lume, "What does a load balancer do?," Lume, [Online]. Available: <https://lumecloud.com/what-does-a-load-balancer-do/>. [Accessed 11 4 2023].
- [17] Plesk, "HTTP/2," Plesk, [Online]. Available: <https://www.plesk.com/wiki/http2/>. [Accessed 10 6 2023].
- [18] srilathaturlapati, "Types of Multiplexing in Data Communications," GeeksForGeeks, [Online]. Available: <https://www.geeksforgeeks.org/types-of-multiplexing-in-data-communications/>. [Accessed 10 3 2023].
- [19] f5, "What is SSL Offloading?," f5, [Online]. Available: <https://www.f5.com/glossary/ssl-offloading>. [Accessed 6 10 2023].
- [20] A. Isaiah, "Caching In Node.js Applications," HoneyBadger, [Online]. Available: <https://www.honeybadger.io/blog/nodejs-caching/>. [Accessed 05 05 2023].
- [21] U. Mansoor, "Caching Strategies and How to Choose the Right One," Codeahoy, [Online]. Available: <https://codeahoy.com/2017/08/11/caching-strategies-and-how-to-choose-the-right-one/>. [Accessed 5 5 2023].
- [22] rromoff, "From streaming to studio: The evolution of Node.js at Netflix," OpenJS, [Online]. Available: <https://openjsf.org/blog/2020/09/24/from-streaming-to-studio-the-evolution-of-node-js-at-netflix/>. [Accessed 12 5 2023].
- [23] "UNDER THE HOOD OF UBER: THE TECH STACK AND SOFTWARE ARCHITECTURE," The app solutions, [Online]. Available: <https://theappsolutions.com/blog/development/uber-tech-stack/>. [Accessed 12 5 2023].
- [24] S. Esemé, "10 Most Popular Types of Node.js Apps in 2023," Kinsta, [Online]. Available: <https://kinsta.com/blog/node-js-apps/>. [Accessed 12 5 2023].
- [25] J. Romanowski, "Which Major Companies Use PostgreSQL? What Do They Use It for?," Learnsql, [Online]. Available: <https://learnsql.com/blog/companies-that-use-postgresql-in-business/>. [Accessed 12 5 2023].
- [26] "What Is Session Persistence?," Nginx, [Online]. Available: <https://www.nginx.com/resources/glossary/session-persistence/>. [Accessed 26 5 2023].
- [27] B. Assmann, "Load Balancing, Affinity, Persistence, Sticky Sessions: What You Need to Know," Haproxy, [Online]. Available: <https://www.haproxy.com/blog/load-balancing-affinity-persistence-sticky-sessions-what-you-need-to-know/>. [Accessed 26 5 2023].
- [28] T. Rascia, "CRUD REST API with Node.js, Express, and PostgreSQL," LogRocket, [Online]. Available: <https://blog.logrocket.com/crud-rest-api-node-js-express-postgresql/>. [Accessed 26 5 2023].

- [29] M. Owolabi, "How To Setup Caching in Node.js using Redis," Hackernoon, [Online]. Available: <https://hackernoon.com/how-to-setup-caching-in-nodejs-using-redis-tq143usy>. [Accessed 26 5 2023].
- [30] G. Callaghan, "Quick guide to load testing with Siege," Medium, [Online]. Available: <https://medium.com/@guy.callaghan/load-testing-with-siege-3fa6a1e8118d>. [Accessed 16 05 2023].
- [31] braktim99, "How to create load balancing servers using Node.js ?," Geeksforgeeks, [Online]. Available: <https://www.geeksforgeeks.org/how-to-create-load-balancing-servers-using-node-js/>. [Accessed 26 5 2023].
- [32] E2logy, "How Is Node JS Used in Web Development?," E2logy, [Online]. Available: <https://e2logy.com/blog/how-is-node-js-used-in-web-development/>. [Accessed 31 03 2023].
- [33] Techtic, "How Node.js Apps made these 16 Companies to Flourish?," Techtic, [Online]. Available: <https://www.techtic.com/blog/companies-use-nodejs-in-web-app/>. [Accessed 12 4 2023].
- [34] NitroPack, "Cache Invalidation - What Is It and Does Your Website Need It?," NitroPack, [Online]. Available: <https://nitropack.io/blog/post/cache-invalidation-nitropack>. [Accessed 13 04 2023].
- [35] Section, "Understanding the Downsides of Load Balancing," Section, [Online]. Available: <https://www.section.io/engineering-education/understanding-the-downsides-of-load-balancing/>. [Accessed 13 04 2023].
- [36] f5, "Cookies, Sessions, and Persistence," f5, [Online]. Available: <https://www.f5.com/resources/white-papers/cookies-sessions-and-persistence>. [Accessed 13 04 2023].

